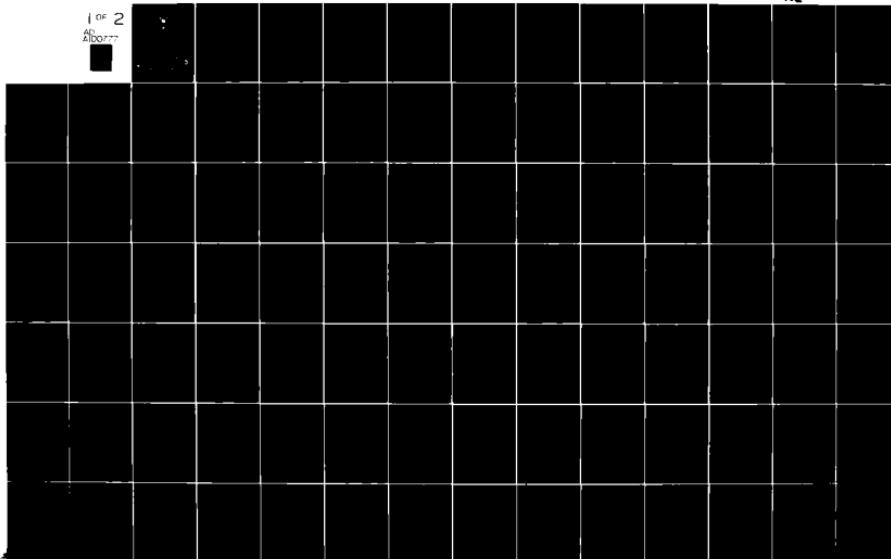


AD-A100 777 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL--ETC F/G 9/2
WRITING A SOFTWARE PACKAGE FOR AN ENR 760 BDC ON THE VAX-11/780--ETC (1)
DEC 80 D L RALL
UNCLASSIFIED AFIT/GCS/EE/800-13

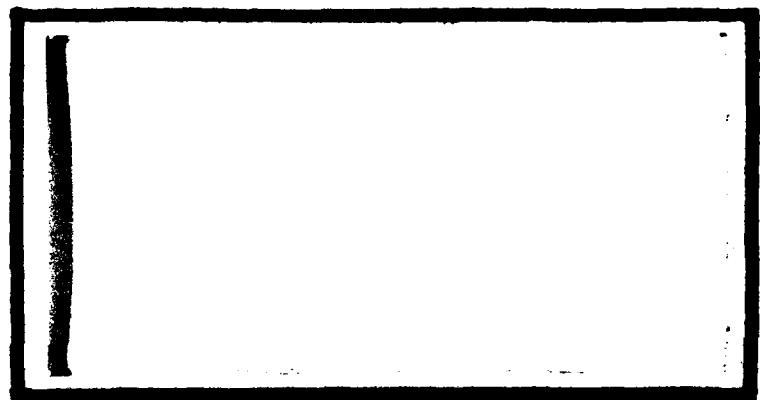
ML

1 of 2
AD-A100 777



00c

AD A100772



DTIC
SELECTED
S JUL 1 1981 D

DTIC FILE COPY

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

A

Wright-Patterson Air Force Base, Ohio

81 6 30 075

AFIT/GCS/EE/80D-13

WRITING A SOFTWARE PACKAGE FOR
AN EMR 760 BDC ON THE VAX-11/780

AFIT/GCS/EE/80D-13 DAVID L. RALL
2LT USAF

3 JUL 1 1981 0

A

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION
UNLIMITED

AFIT/GCS/EE/80D-13

WRITING A SOFTWARE PACKAGE FOR AN EMR 760 BDC ON A VAX-11/780

THEESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

in Partial Fulfillment of the
Requirements of the Degree of

Master of Science

By
David L. Rall, B.S.
2nd Lieutenant USAF

Graduate Computer Systems

December 1980

Association For
ENTERTAINMENT
PROFESSIONALS
IN THE FIELD
OF ENTERTAINMENT
AND RELATED INDUSTRIES

Approved for public release: distribution unlimited

Preface

A data acquisition computer at the Air Force Weapons Laboratory is in the process of being replaced by a VAX-11/780. The hardware interface needed to connect their telemetry equipment to the VAX-11/780 was built by Electro-Mechanical Research. No software support is available for the interface, Electro-Mechanical Research Model 760 Buffered Data Channel. This report describes the software package designed to begin transferring the data acquisition workload to the VAX-11/780. This software package is intended only as a building block for the computer professionals at the Air Force Weapons Laboratory.

Special thanks are due to several technicians at the Air Force Weapons Laboratory who made my two week stay at Kirtland very rewarding. Thanks are due to Ralph Warrmack, who provided the humor and hardware expertise needed to perform adequate testing on the device driver, and Gene Simpson, who provided assistance and advise in making necessary refinements in the application of the driver. Thanks are also due to Ken Summers, who provided the initial direction in writing the device driver and also assisted in the testing of the driver. A very special thanks are due to Lt Randy Rushe. On several occasions, Lt Rushe provided the answers necessary to overcome the major obstacles involved with loading a device driver.

Thanks are also due to my theses advisor, Prof Gary La-

mont, for the continual advise needed throughout the writing of this report.

A very special thanks are in order to my wife, Vicki, who had to endure the critical months of her first pregnancy alone.

David L. Rall

List of Figures

Figure	Page
1. Current System	3
2. Interface to UNIBUS	3
3. Final Configuration	5
4. Initial Configuration	5
5. Single Mode Use of EMR 760	12
6. MAX, MA, and WC registers	12
7. Single Mode: 4 Buffers (65KW) ...	13
8. Command Register 1	121
9. Command Register 2	123
10. FIFO Refill	16
11. Adaptive Burst Mode	18
12. Telemetry Equipment	23
13. One Buffer System	23
14. Double Buffer System	23
15. Page Tables	34
16. VAX-11/780 Hardware Configuration	34
17. UNIBUS to SBI Address Translation	37
18. SBI to UNIBUS Address Translation	40
19. Map Register Bit Configuration ..	40
20. Processing a DMA Read Operation .	59
21. Initialization Routine	66
22. EXE\$READ Routine	68
23. Start I/O Routine: Initialize ...	71

Figure	Page
24. Interrupt Routine 1: Block End ..	74
25. Interrupt Routine 2: End Transfer	74
26. Start I/O Routine: Completion ...	76
27. Initial User Program	83
28. Continuous Sample Data	89
29. Application Program	91
30. Front End Microprocessor	98
31. Network Configuration	98

List of Tables

Table		Page
I.	UNIBUS Field to SBI Field Translation	42
II.	SBI Function-Mask Translation	43
III.	UNIBUS Device Address Space	42
IV.	Verification of Transfer Rate	95

Contents

Preface	ii
List of Figures	iv
List of Tables	vi
Abstract	ix
I. Introduction	1
Background	1
Problem	4
Summary	6
Approach	6
Thesis Development	7
II. Requirements	9
EMR 760 BDC	9
Telemetry Equipment	20
Selection of Operating Modes	22
VAX-11/780	30
Summary	46
III. Development of Device Drivers	48
Device Driver	48
Fork Process	50
Interrupt Priority Level	51
Fork Queue	53
Resource Wait Queue	53
I/O Data Base	54
Control Blocks	55
I/O Request Packets	57
Overview of a DMA Read Operation	58
Summary	62
IV. Design of an EMR 760 Driver	64
Controller Initialization Routine	65
Function Decision Table Routine	65
Start I/O Routine	69
Interrupt Service Routine	73
Reactivation of Start I/O Routine	75
Alternatives	77

V.	Verification and Validation	80
	Syntactic Errors	80
	Walkthrough	80
	Loading the Driver	81
	Testing the Driver	82
	XDELTA	84
	Selection of Data Paths	85
	Interrupt Handling	85
	Simulation	88
	Application Program	88
	Transfer Rates	92
	Success	93
	Summary	94
VI.	Conclusions and Recommendations	96
	Integration	96
	Transfer Rates	99
	Degradation of System	101
	Application Program	101
	Summary	103
	Bibliography	105
	Appendix A: EMR 760 Driver	106
	Appendix B: User Program	114
	Appendix C: Figures 8 and 9	121
	Vita	125

Abstract

The Air Force supports many research laboratories and test centers. These laboratories and test centers collect information on simulations and tests performed at their facilities. The main medium for collection of this data is analog tapes. The analog tapes are forwarded to data acquisition centers that process the data.

The data acquisition center at the Air Force Weapons Laboratory (AFWL) is upgrading their computer system. A VAX-11/780 computer will replace the obsolete UNIVAC 6135 computer. An Electro-Mechanical Research (EMR) model 760 Buffered Data Channel (BDC) will be used to interface their existing telemetry equipment (analog tape drives, A/D converters, time code converters) to the VAX-11/780.

A device driver for an EMR Model 760 Buffered Data Channel was written and tested for the VAX-11/780. The concepts involved with writing a device driver are discussed. An application program, using the device driver, was written and evaluated on the basis of overall system performance. Transfer rates in excess of 190,000 words per second were maintained over the UNIBUS. The EMR 760 successfully transferred over 10,000,000 bytes of data to a file on an RP06 disk. The features of the VAX-11/780 that concern device drivers are presented.

I. Introduction

The Air Force supports many research laboratories and test centers. These laboratories and test centers collect information on simulations and test performed at their facilities (in flight analysis, wind tunnel tests, equipment stress, etc). Analog tapes are used to collect this information and are forwarded to conversion centers that digitize and process the data.

The data conversion center at the Air Force Weapons Laboratory (AFWL) is in charge of data acquisition for AFWL and the surrounding labs and test centers. Analog tapes are forwarded to the data conversion branch by AFWL and supporting labs. It is the data conversion branch's responsibility to digitize the data and collect portions of the data for further processing. The data conversion branch uses expensive telemetry equipment to digitize the information and collect only those regions specified by the supporting lab. After the data has been collected, it is processed according to the user needs, and then sent to forwarding labs for extensive analysis.

Background

The data conversion branch is currently maintaining their workload on a 10-15 year old EMR-UNIVAC 6135 computer.

Figure 1 depicts the computer system, with telemetry equipment, that is currently being used to digitize, collect, and process the data. The telemetry equipment (Figure 1) consists of analog tape drives, time code converters, filters, and A/D converters. The EMR-UNIVAC 6135 is a special purpose computer built solely for this data acquisition environment. However, the computer is over ten years old and will no longer be maintained by UNIVAC after January 2 of 1982. Without a maintenance contract, the computer would not be very reliable. Since the EMR-UNIVAC 6135 was such a special purpose machine, not many were marketed and the number of spare parts would be minimal. The only appropriate solution was to buy a modern computer system that would provide the same abilities as the 6135 while allowing an expansion of their current facilities.

Electro-Mechanical Research (EMR) built several 6135 computer systems that were used with telemetry equipment similar to that at AFWL. Knowing they would no longer be able to maintain the 6135, EMR built an interface for the telemetry equipment. This interface, the model 760 buffered data channel, provided a means to connect the telemetry equipment to a different bus structure than that of the 6135. The EMR 760 allows the telemetry equipment to be interfaced to a UNIBUS structure (Figure 2). There are several bus structures similar to the UNIBUS, but Digital Equipment Corporation (DEC) provides that exact bus structure on its PDP-11 family of computers. DEC has recently expanded

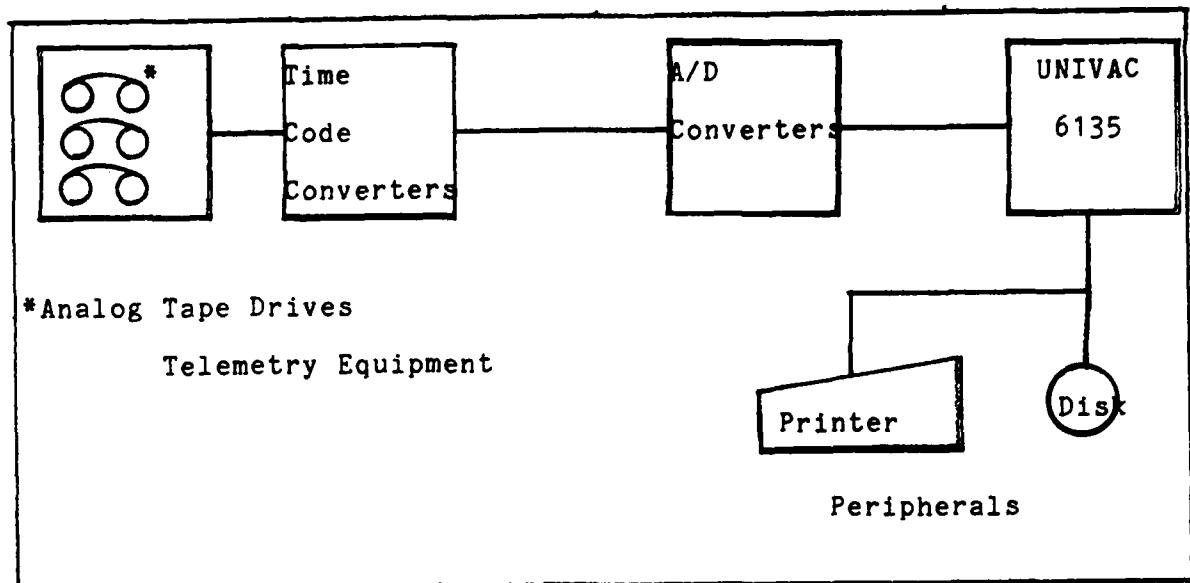


Figure 1. Current System

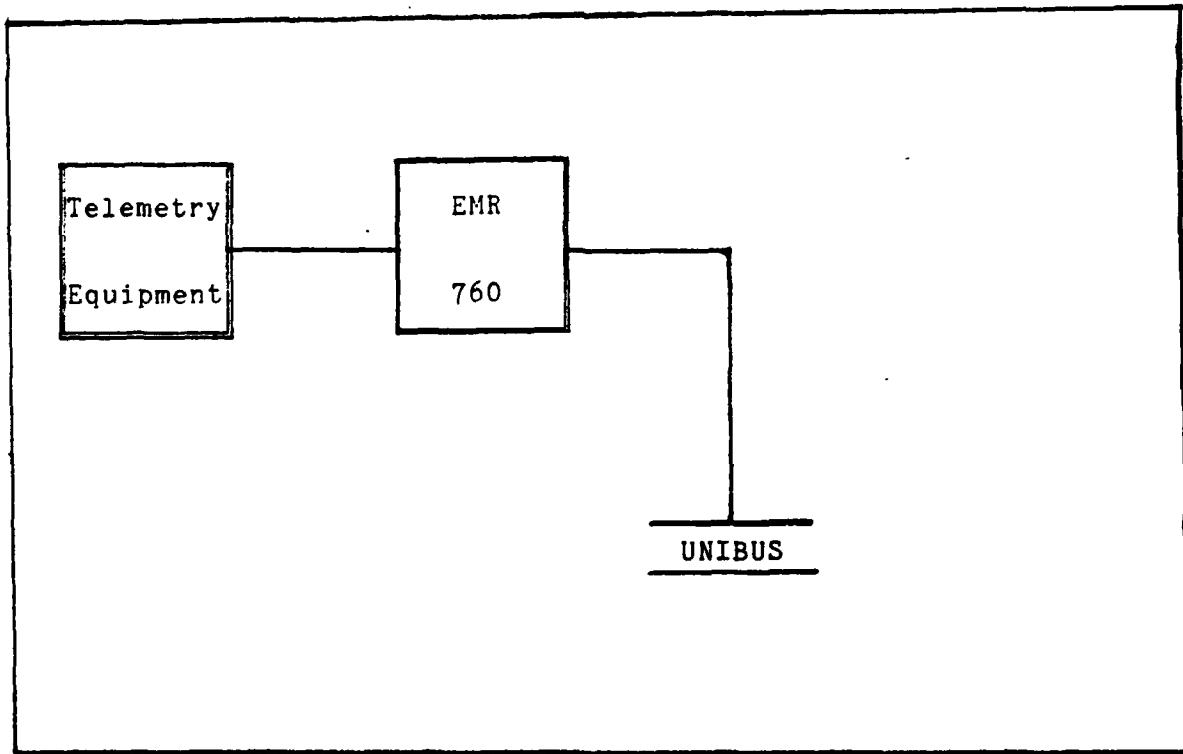


Figure 2. Interface to UNIBUS

its PDP-11 series of computers and the VAX-11/780 is the product of that expansion (Ref 2:vii).

The data conversion branch at AFWL purchased a VAX-11/780, and will be integrating the VAX-11/780 into their system. The final configuration of this system will be similar to that shown in Figure 3. For an initial building block, the data conversion branch would like to start out with a system depicted in Figure 4. The problem arises with the software support for the EMR model 760: there is none. The model 760 has been on the market for a very short time and EMR has not had time to develop software support for the model 760 on a VAX-11/780.

Problem

Software needs to be developed to support the EMR 760 on a VAX-11/780. This software should include a device driver and a user program. The device driver should initialize device registers, handle interrupts requested by the device, and any other operations associated with the device. The user program will thus allocate buffer areas, create a file on disk, initiate a DMA transfer between the EMR 760 and computer memory, and transfer data from computer memory to the file on disk.

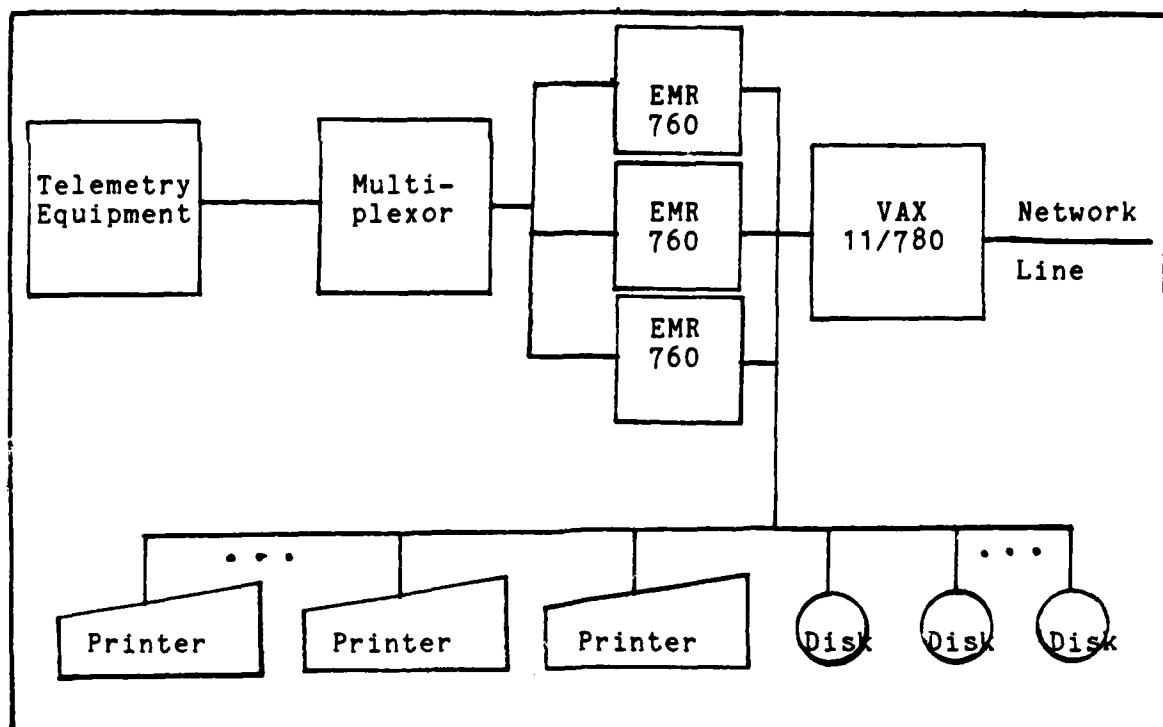


Figure 3. Final Configuration

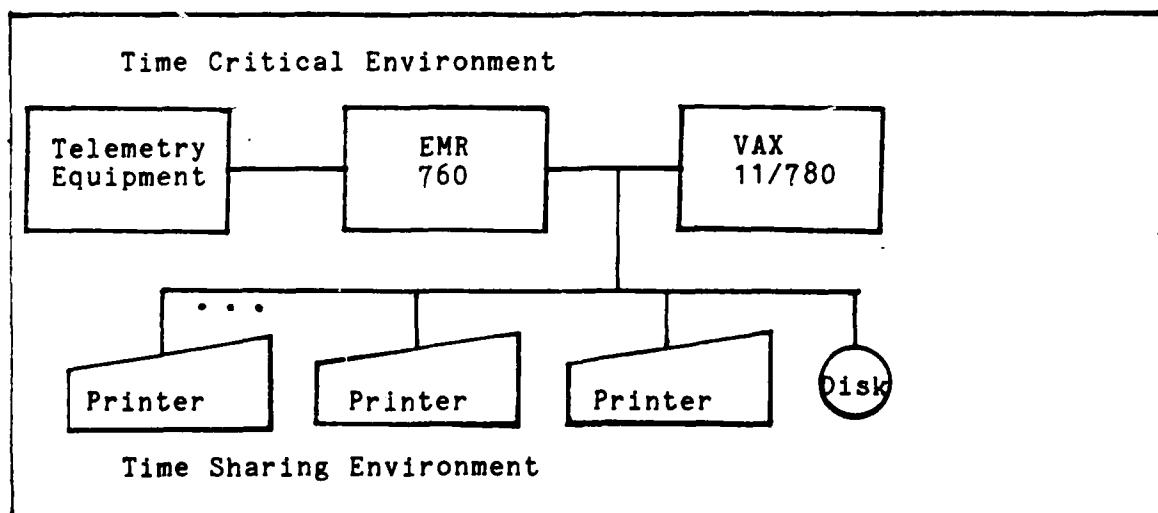


Figure 4. Initial Configuration

Summary of Current Knowledge

The VAX-11/780 and the EMR model 760 were the main considerations in this research and the only knowledge of either system comes from the vendor supplied manuals. For the VAX-11, this consists of a series of system manuals that are supplied with every system. There are also handbooks (Refs 1, 2, and 3) that can be obtained at any Digital Equipment Corporation office. These handbooks were very helpful throughout the research and are highly recommended as supplementary reading for this report. The documentation for the EMR 760 consists of a user manual (Ref 4) describing the operation of the EMR 760.

Approach

Because of the complexity of the VAX-11/780, the majority of information in this report is from research of the system routines, the assembly language, the architecture, and the VAX Virtual Memory System (VAX/VMS). Two software programs were designed and coded. One program is the driver for the EMR 760 and the other program performs the necessary processing on the data while it is transferred to computer memory (this processing will be to simply transfer the data to disk while it is being transferred from the EMR 760).

The driver was assembled, debugged, and loaded into the

system at AFWL. Afterwards, extensive testing based upon the transfer to computer memory was performed. This was done to insure the integrity of the DMA transfer.

The main objective of this investigation was to begin the transition from the UNIVAC 6135 to the VAX-11/780. This includes a driver for a single EMR 760 and a corresponding application program. It was not feasible to provide a multiunit controller. It would take six months or more to develop such a sophisticated driver and controller. The personnel at the data conversion center were more interested in the initial step of transferring data to computer memory. For this reason, most of the emphasis is on the development of the device driver.

Thesis Development

The development of this thesis proceeds in five phases. In the first phase (chapter II), the requirements and constraints of the equipment in the system will be discussed. This will include introductory information on the EMR 760 and the VAX-11 system. The second phase (chapter III) will address the issue of a device driver and the tables and routines needed by the driver. The third phase (chapter IV) will discuss the design of a driver for the EMR 760, and will list the alternatives that could have been taken. The fourth phase (chapter V) will discuss the procedure followed

to load and debug a device driver in a VAX-11 system. There will be a brief discussion of the transfer rate achieved by the system, but the transfer rate was a secondary issue. The final chapter will cover conclusions and recommendations, with a brief discussion of on-going thesis research that could be performed at AFWL. Appendix A and B contain the code for the device driver and user program, respectively.

II. Requirements

This chapter defines the requirements and the AFWL system environment of the proposed configuration. This includes discussion of the EMR 760, the telemetry equipment, and the VAX-11/780. All information provided on the EMR 760 has been taken from Ref 4. If a hardware or electrical description is needed of the EMR 760, the reader is referred to Ref 4. The only source of information on the telemetry equipment was provided by verbal communication with the technicians at AFWL. The information on the VAX-11/780 was taken from three sources: Ref 2, Ref 3, and Ref 1. These three sources provide a comprehensive discussion of the VAX-11 system and are highly recommended as prerequisites before reading this report.

EMR 760 Buffered Data Channel

This section discusses the EMR 760 and should provide a general idea of how AFWL will be using the EMR 760 to begin their data conversion. The telemetry equipment will provide the input to the EMR 760 and a program will provide the necessary control over the initialization of the EMR 760. The telemetry equipment will not utilize every mode of the EMR

760, but the different modes it can use will be discussed. The EMR (Electro-Mechanical Research) 760 Buffered Data Channel (BDC) is a high-speed, DMA (Direct Memory Access) device that transfers 16-bit words to or from computer memory. The data conversion branch at AFWL will be using several EMR 760's in their conversion from an EMR-UNIVAC 6135 computer to a DEC VAX-11/780 computer. Using several EMR 760s will provide a greater throughput rate but will increase the complexity of the controller. Only one EMR 760 was used in this research. The EMR 760 provides the hardware interface needed to connect their telemetry equipment to the UNIBUS of the VAX-11/780. The EMR 760 provides several operating environments including a single buffer or cyclic buffer input to computer memory. Although the EMR does offer an output mode, it will not be used by AFWL and, therefore will not be discussed.

Single Buffer. In the single buffer mode the EMR 760 transfers data from an external device, the telemetry equipment, to one, two, three, or four buffers in computer memory. Each buffer can have up to a maximum of 65K words (Figure 5). The EMR 760 provides four Memory-Address (MA) and four Word-Count (WC) registers. The MA registers have a corresponding Memory Address Extension (MAX) register and together these registers provide the 18-bit address needed to address a location in the UNIBUS address space. The 16-bit WC registers provide the total word length of their corresponding memory buffers. The concept of the MA, MAX, and WC

registers in relation to their buffers in computer memory is illustrated in Figure 6. These registers are set-up by the computer before the EMR 760 is activated. Upon activation, via the command (control) registers, the EMR 760 transfers the first MA, MAX, and WC registers to the Memory Address Counter (MA CTR) and the Word Count Counter (WC CTR). The data is now transferred to the memory buffers and after each transfer of one word, the MA CTR is incremented and the WC CTR is decremented. This continues until the WC CTR is decremented to zero. When this occurs, the next MA, MAX, and WC registers are transferred to the MA CTR and WC CTR. The operation is repeated for as many blocks as specified in the command (control) registers (Figures 8 and 9; because of the length of Figures 8 and 9, they are provided as Appendix C). Figure 7 describes the sequence of events when the single buffer mode is used. After the EMR 760 has transferred data to the full range specified, it will generate a block end signal. The block end signal stops the EMR 760 and if interrupts are enabled for the device, the EMR 760 will generate an interrupt on the UNIBUS. The block end signal also signifies the completion of the single buffer operation. This provides for a limited number of data input to the computer. For continuous input to the computer the EMR 760 provides a cyclic buffer input.

Cyclic Buffer. The EMR 760 provides a cyclic mode operation for data streams of undetermined length. In cyclic mode the EMR 760 transfers data to four memory buffers in

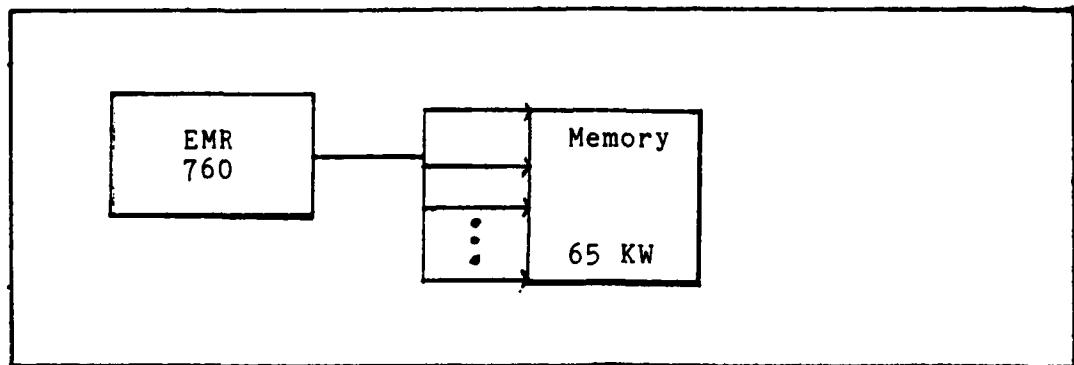


Figure 5. Single Mode Use of EMR 760

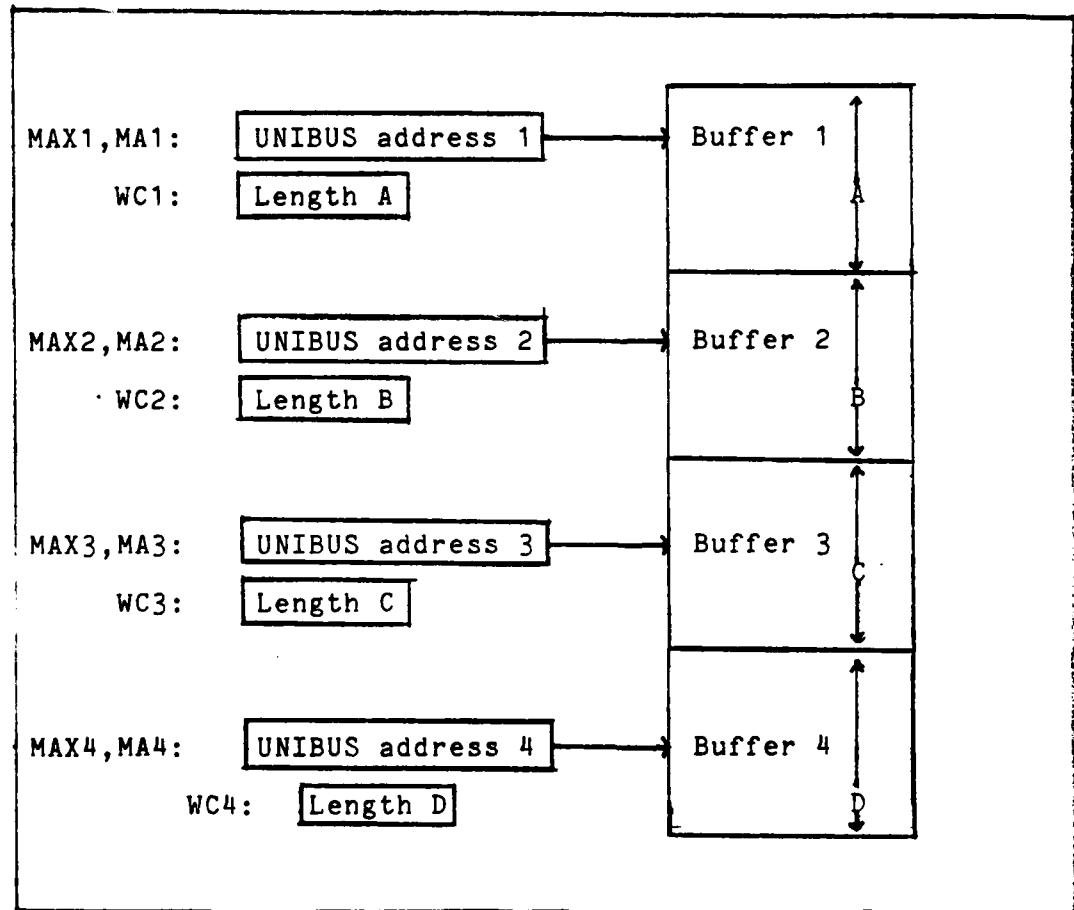


Figure 6. MAX, MA, and WC registers

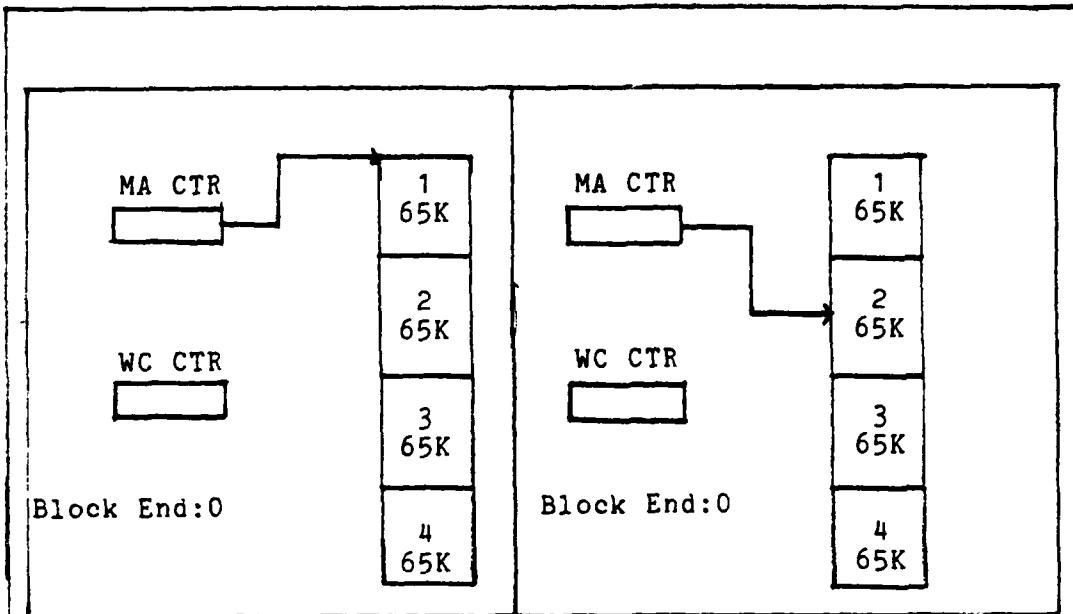


Figure 7.a Start of Transfer Figure 7.b In Buffer 2

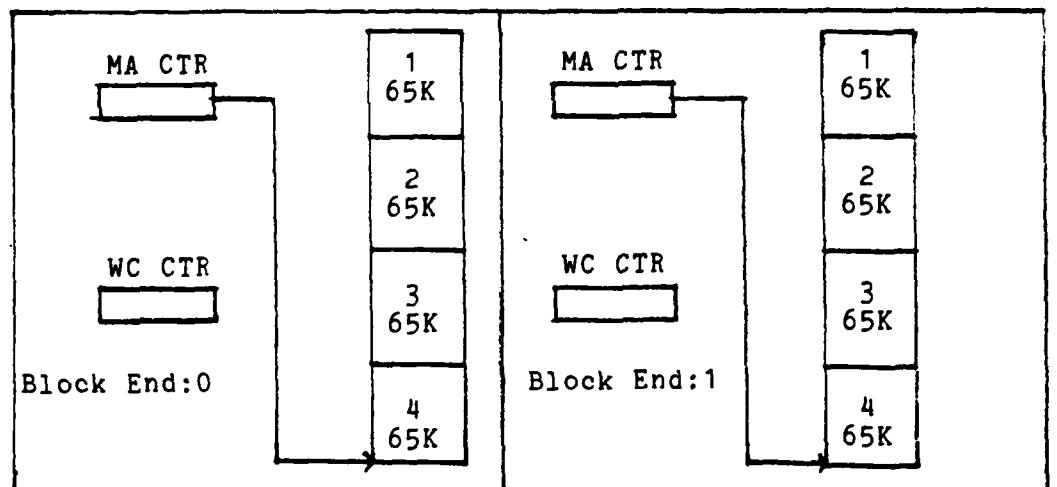


Figure 7.c Last Word

Figure 7.d Completion

Figure 7. Single Mode: 4 Buffers (65KW)

the same manner as described above. The only exception is that the block end signal is not generated after the final memory buffer is filled. Instead, the EMR 760 transfers the first MA-WC pair to the MA CTR and WC CTR and continues in the cyclic fashion until either the program or an external signal deactivates the EMR 760, via the control registers (Figures 8 and 9). In cyclic mode, DMA operations continue unrestricted by the EMR 760 (Figures 8 and 9).

Interrupts. The EMR 760 provides two interrupts for controlling the operation of the EMR 760. The first interrupt is dedicated to the block end signal and is controlled by the command registers (Figure 8). When bit 8 of control register 1 is set (1) and the EMR 760 is operating in non-cyclic mode, interrupt 1 will be requested on the UNIBUS after the completion of data transferred to the final buffer. Interrupt 2 is a general purpose interrupt and can be assigned to signal a FIFO overflow or user defined event. If bit 9 of control register 1 is set, interrupt 2 will be requested on the UNIBUS when either the FIFO overflows or a user input signal was detected, depending on bit 10 of control register 1. If bit 10 of control register 1 is set (1), interrupt 2 will be dedicated to signal an overflow of the FIFO. The FIFO is a First-In First-Out buffer and can be used to store up to 16 data words while the EMR 760 is preparing to burst them over the UNIBUS.

Bursting. The EMR 760 provides two modes of bursting

data from the device to the UNIBUS: regular or adaptive ("hog") burst. Bursting is provided to minimize the time needed for UNIBUS arbitration. In burst mode the EMR 760 stores up to fourteen 16-bit words in a FIFO buffer area. When the number of words, as requested by bits 12-14 of control register 1, is accumulated; the EMR 760 requests control of the UNIBUS and does not release it until the FIFO is emptied. This decreases the number of times the UNIBUS needs to be requested, thereby decreasing the overall time used during a transfer session. The sequence of events described above are considered the regular burst mode of the EMR 760. But, what about the words that were transferred to the FIFO while the EMR 760 was transferring the FIFO to the UNIBUS (Figure 10)? If the EMR 760 has 14 words to transfer to the UNIBUS, obviously the input data will already have begun to accumulate in the FIFO again. The adaptive burst mode utilizes these words to transfer more than the specified number of words in the FIFO.

The adaptive burst mode maintains control of the UNIBUS until the entire FIFO buffer is emptied. Figure 10 demonstrated how lower locations of the FIFO were being filled even while the upper locations of the FIFO were being emptied. Figure 11 describes the sequence performed by the EMR 760 to empty the entire contents of the FIFO (even the words that were filled after the operation was started). There are two locations, A and B, which will be used to simplify the explanation of this technique. Location A contains the

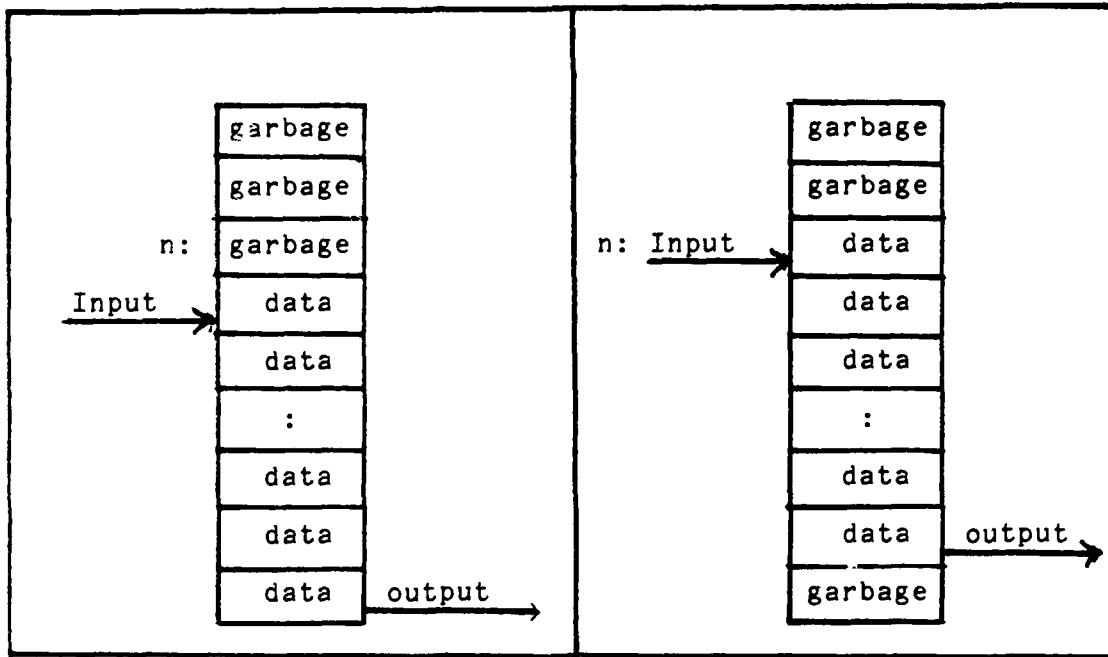


Figure 10.a Initial Fill

Figure 10.b FIFO Full

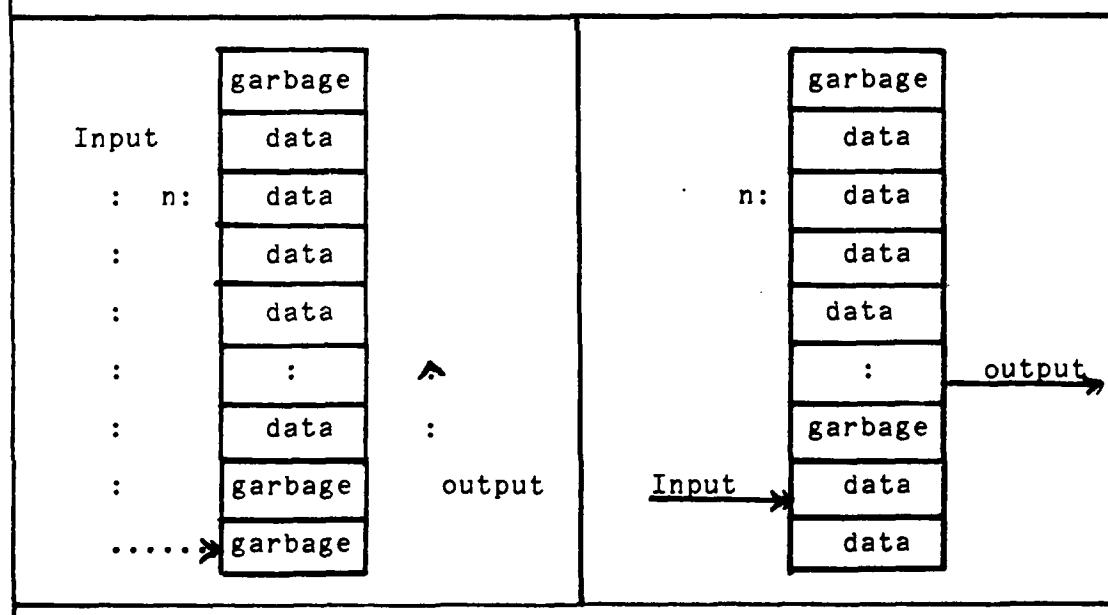


Figure 10.c Flushing FIFO

Figure 10.d Refilling FIFO

Figure 10. FIFO Refill

number of words that have been stored in the FIFO, initially it is zero. Location B contains the number of words that the EMR 760 has transferred out of the FIFO buffer, it is also initially zero (Figure 11.a). When a word of data is stored in the FIFO, location A is incremented by 1. When A becomes equal to the number of words requested, the EMR 760 begins sending data (Figure 11.b). While the EMR is sending the first word to the UNIBUS, the next input word is stored in location A+1 of the FIFO. Location A is then set to zero. Location B has been maintaining its count while this was happening. The input data is now starting to be filled into the lower locations of the FIFO. When B becomes equal to the number of words requested to be stored in the FIFO plus one, it is reinitialized to zero. Data continues to be transferred from the FIFO to the UNIBUS until B equals A (Figure 11.d). Now everything is reinitialized and the UNIBUS is released. The process is guaranteed to stop because the EMR 760 outputs data to the UNIBUS two to three times faster than it allows input into the FIFO. The adaptive burst mode is the most efficient method of operating the EMR 760 and can be set by bit 11 of control register one (Figure 8). Another bit that deserves some comment is the XST (external start) which is bit 7 of control register 1 (Figure 8).

External Start There are two methods of activating the EMR 760. The first is considered the direct activation of the EMR 760 and will be used when the XST bit of control re-

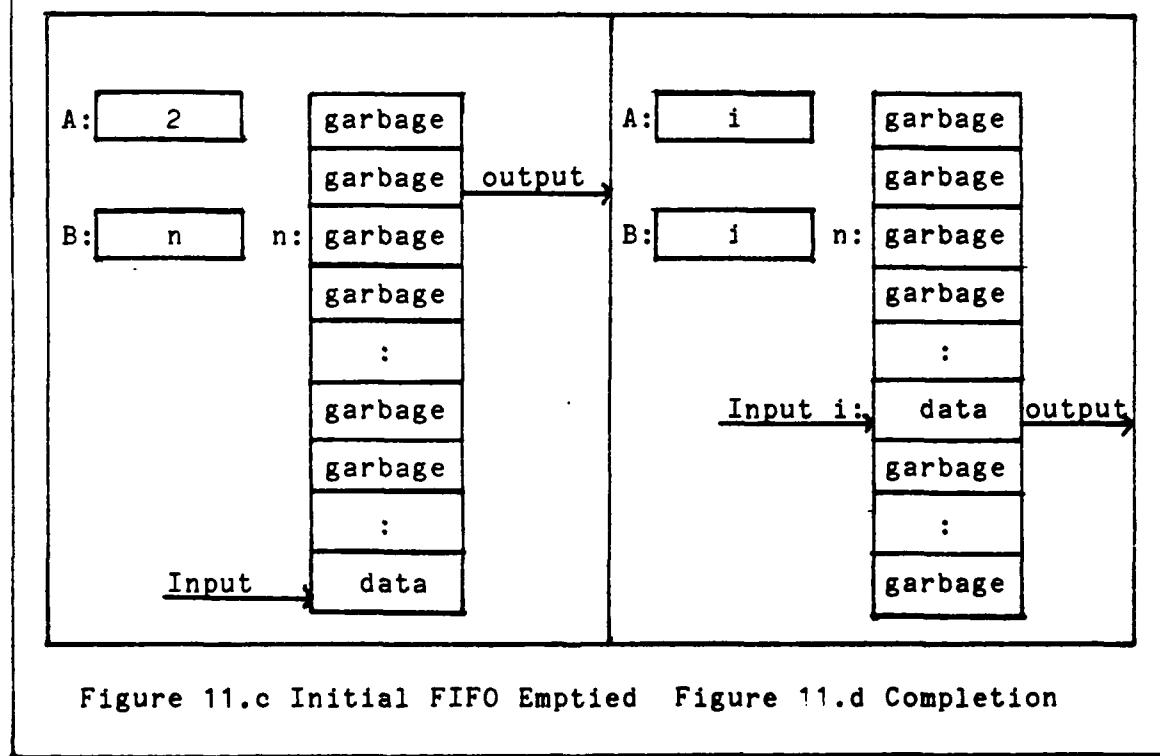
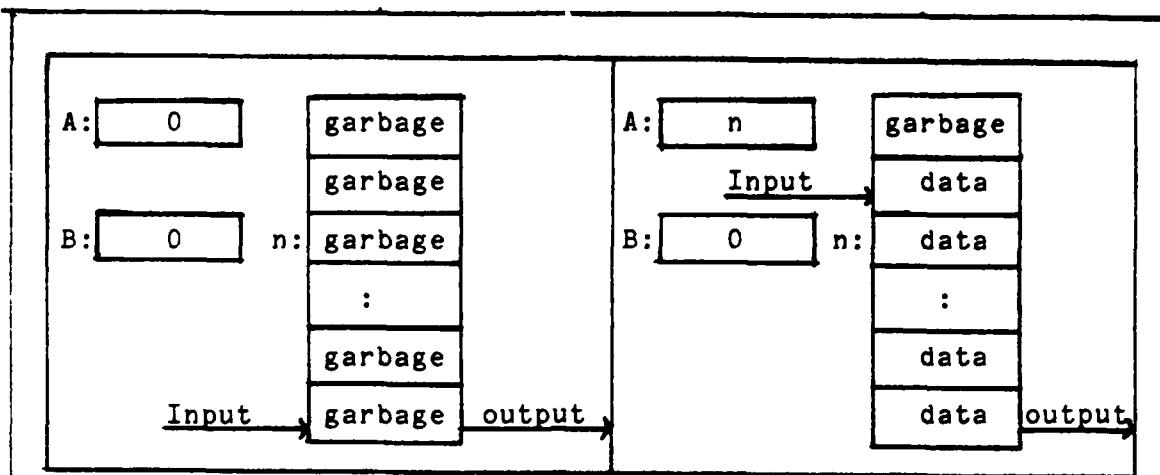


Figure 11. Adaptive Burst Mode

gister 1 is cleared (0). With the XST bit cleared, the EMR 760 is activated by setting the ON bit of control register 1 (Figure 8). All device register initialization will be done before the ON bit of register 1 is set (1). After the ON bit has been set the EMR 760 awaits input from the external user device, the telemetry equipment. The EMR 760 continues transferring data until the ON bit is cleared (0). This can be accomplished via a block end signal or by one of the two interrupts. This mode of operation is performed when the XST bit is cleared (0).

If the XST bit is used in conjunction with the ON bit, the user can perform a delayed start of the EMR 760. If the XST bit is set (1) and the ON bit is set (1), the EMR 760 is considered "armed". In the armed state the EMR 760 will not transfer data. Only after this XST bit is cleared will the EMR 760 be allowed to transfer data. This can be performed by either program control or a user signal wired to that bit. If a user signal is wired to that bit, the XST can be cleared by simply grounding the user line. Grounding of the user signal will usually occur when the user wants to begin transferring data to the EMR 760. Once the XST bit is cleared, provided the ON bit is still set (1), the EMR 760 awaits input from the external user device, the telemetry equipment. The EMR 760 continues transferring data until the ON bit is cleared by either method discussed earlier.

This completes the discussion of the EMR 760 and should

have given the reader an idea of how AFWL will be using the EMR 760 to begin their conversion from the EMR-UNIVAC 6135 to the VAX-11/780. The telemetry equipment will provide the input to the EMR 760 and a program will provide the necessary control over the initialization of the EMR 760. The telemetry equipment will not, obviously, utilize every mode of the EMR 760. The discussion above showed the different operating modes of the EMR 760 and the discussion of the telemetry equipment in the next section will define the modes needed for the interface of the telemetry equipment to the EMR 760.

Telemetry Equipment

The telemetry equipment provide the data conversion branch at AFWL with the necessary hardware support needed to complete the initial phase of the data aquisition cycle. The telemetry equipment consists of many analog tape drives, time code converters, and A/D converters. Figure 12 illustrates the utilization of the telemetry equipment and the configuration of that equipment at AFWL. Each portion of the system is discussed in greater detail in the following sections:

Analog Tape Drives. Analog tape drives are used to read data from analog tapes. The analog tapes are provided by many of the labs in AFWL's surrounding area. These tapes

are recorded for many reasons: simulation data, in flight testing, ground turbulence data, etc.. An analog tape may consist of millions of bits of information. Only a small portion of that information might be needed, but the portion that is needed must be transferred continuously, without interruption. This is the very nature of analog tapes; they cannot be stopped and started in the same place. Therefore, this small portion of data, say 2-5 megabytes (digital representation), has to be detected and then sent to the computer in a continuous stream of data!

Time Code Converters. The detection of the start of information is provided by time code converters. One requirement specified by the data conversion branch at AFWL was that analog tapes that needed to be processed had to have time codes provided in the data. The time interval of these time codes could be one tenth of a second or even a second in length. The lab requesting processing would also specify the time frame of the data it needed, say 20.3 seconds to 24.7 seconds. With this information, the time code converters would know precisely when to begin transferring data and precisely when to stop. The analog data that needs to be transferred to the computer needs to be digitized: that is the function of the A/D converters.

A/D Converters. The A/D converters digitize incoming analog data. This digitized form can then be transferred to the computer. Many computers have built in A/D converters

but, unfortunately, are never of the caliber needed by high-speed telemetry equipment. This necessitates the need for external A/D converters and that is why they are utilized in this configuration. The digitized form of data from the A/D converters is provided as input to the EMR 760. This data arrives as a continuous stream and cannot be interrupted. The time code converters provide the necessary control for the EMR 760 and signals it when a stream of data is being prepared for transfer. The next section discusses the selection of operating modes for the EMR 760 based upon the needs of the telemetry equipment.

Selection of Operating Modes

There are several operating modes of the EMR 760 which could satisfy the criteria set forth by the telemetry equipment. Each mode will be discussed and some thought will be given to the desired EMR 760 setup and the initial setup provided by this research.

Buffer Selection. The selection between the single buffer mode and the cyclic buffer mode is straightforward. Since the information is a continuous data stream of an undetermined length, the cyclic mode will have to be used. This will allow the EMR 760 to continuously transfer data to one, two, three, or four buffers for an unlimited amount of time. The selection of the number of buffers and the buffer

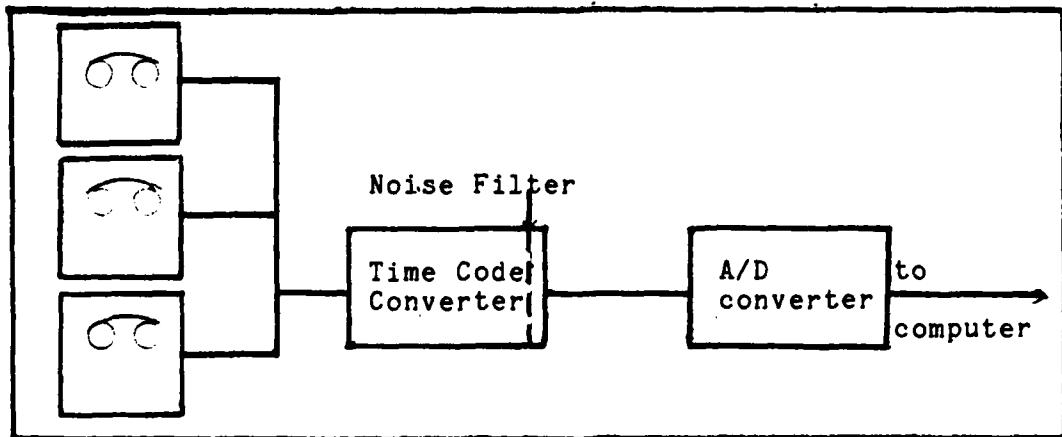


Figure 12. Telemetry Equipment

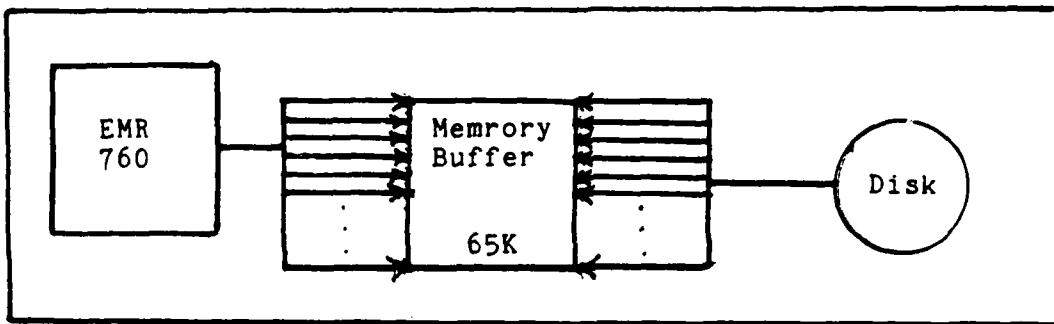


Figure 13. One Buffer System

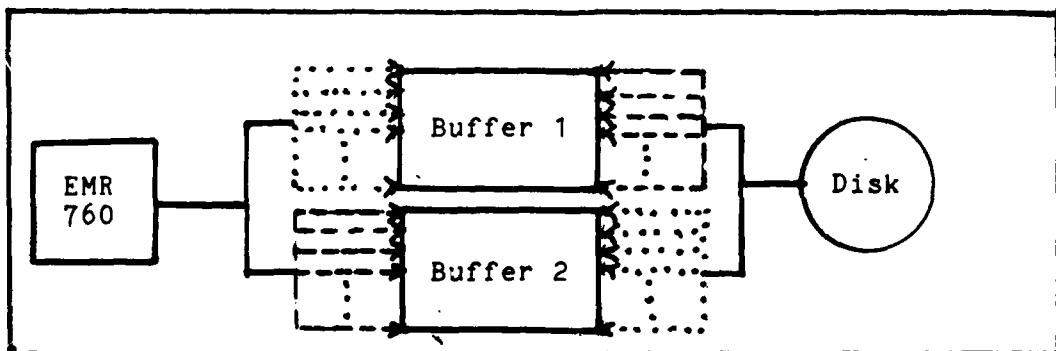


Figure 14. Double Buffer System

length seems not as straightforward as the selection between single and cyclic buffer mode.

To choose the optimal number of buffers for this application the designer must not only consider the viewpoint of the EMR 760 but also that of the disk drive, which will obviously have to be used (the need for a disk transfer will be discussed further and the reader is asked to accept this fact at this time). Transfers to disk can be performed in a block fashion, but the more blocks (buffers) used the more complex the algorithm has to be to control the transfer of those blocks. With this in mind, some discussion on the minimum number of blocks is in order.

The minimum number of blocks the EMR 760 needs to continuously transfer data is one. Figure 13 illustrates what a system utilizing one buffer area would look like. Obviously, the EMR 760 cannot be transferring data to that one block while the disk is trying to read that same block. If this were to occur, the disk, if it were transferring faster than the EMR 760, could at some point go beyond the EMR 760 and transfer bad data and visa versa. The other case would be to stop the EMR 760 while the disk was reading the block, but this cannot be done because the EMR 760 cannot be stopped once started (a restriction from the analog tape drives). The only other alternative would be to have a minimum of two buffers in memory.

With two buffers in memory, the EMR 760 could be

transferring data to one while the disk was reading data from the other (Figure 14). Initially the EMR 760 could start transferring data to buffer 1. As soon as it was finished with that buffer the disk could then store that buffer. When the EMR 760 was finished with buffer 2, the disk would store buffer 2 (Figure 14). It was mentioned earlier that a disk was needed and this will be explained in the next section.

If a disk were not used, the EMR 760 would be transferring data to buffers in computer memory and that data would have to undergo some sort of processing (as established by the contributing laboratory). If the processing of one buffer were to take longer than the time needed for the EMR 760 to fill a buffer, the data could eventually be overwritten. This would have a drastic affect on the output and must be avoided. If the data was transferred to a file on disk, it could be retrieved and processed at a later point in time. This would also allow the EMR 760 to transfer at a rate as fast as the disk. The disk will be transferring data from computer memory to a file. This will take a finite amount of time and if the EMR 760 is transferring faster than the disk, overwriting the data becomes a problem. So, the EMR 760 can transfer as fast as (but not greater than) the disk. This is more desirable than processing the data because the disk transfer will take a constant amount of time, while processing the data could change drastically with different applications. The EMR 760 will still be able

to transfer at a rate as fast as the disk. So, it appears that a configuration of two buffers could satisfy the minimum requirement needed by both the EMR 760 and the disk. It could also be shown that this is the optimum solution because the addition of every new buffer area would increase the complexity and speed of the algorithm that would control the process without any significant advantage to the system as a whole. So, indeed, two buffers will be used. But of what size should they be?

The memory buffer size is limited by two factors: the 16-bit word count register and the number of map registers on the UNIBUS (the UNIBUS map registers will be discussed in detail further in this chapter). The 16-bit word count registers allow the buffer size to be no greater than 65,536 (65K) words. Using two memory buffers this would yeild an effective size of 131,072 (131K) words. There are 496 UNIBUS map registers and each one can address 512 bytes of computer memory. This yeilds 126,976 words of computer memory addressable. Naturally, 126,976 words would be the optimum buffer size because in this instance the larger the buffer, the longer it takes to fill that buffer, thereby giving the disk more time to store the data. This also makes the overhead needed to set up the disk operation have less an affect than it would if a much smaller buffer size was used. But, allocating all the map registers for this operation might take a little time. This is true because other processes, of lower priority, that were using UNIBUS

map registers might be suspended and wouldn't be reactivated until the data aquisition process was finished. In essence, there might be a possible deadlock situation. Therefore, with little justification, only 128 UNIBUS map registers will be used in the data transfer. This yeilds an effective size of 32,768 (32K) words. This will be split equally among the two buffers, allocating 16K words to each. So, a considerable discussion was needed to select just one of the alternatives, but the discussion provided a very realistic overview of the entire process and without further adew the selection criteria will continue.

Interrupts. The selection of the interrupts needed by the system is very simple. The block end interrupt will not be used because the cyclic mode operation does not require it. So, the second interrupt can signify a FIFO overflow or an external user event. Since the FIFO length can be adjusted to almost any size from 2-14 words, the overflow of the FIFO can be controlled. Using the interrupt to signal the end of the transfer session would not only be sensible but, also, easily adapted to the system. The time code converters were used to start the transfer session, so, why not use them to stop the session! The time code converters are given a signal to stop via the time code on the incoming data. It would be very easy to send this stop signal on, to the EMR 760. Indeed, the second interrupt will be used in such a fashion. It was stated that the FIFO could be somewhat controlled and, in fact, the FIFO can be completely

controlled as demonstrated in the following section.

Bursting. There are two modes of bursting: adaptive and regular bursting. These were explained earlier and it was shown that the adaptive burst was the faster. Therefore, it would be desirable to use this mode. In a final configuration utilizing many EMR 760'S the adaptive burst with a burst length of 14 words would definitely be used. In this simple application of one EMR 760 it will not be used. The reason for this is simple and was stated earlier in the introduction. The purpose of this research was to perform the initial step needed to begin the transition from the EMR-UNIVAC 6135 computer to the VAX-11/780. Namely, demonstrate the interface between the telemetry equipment and the VAX-11/780 could be accomplished. The purpose was not to maintain an optimum transfer session over the UNIBUS and, therefore, the simplest method of transfer was used, no bursting at all. This is not to say that bursting shouldn't be part of a desired final configuration. Indeed, it should be. Now a few words about starting the EMR 760.

External Start. As stated earlier, there are two methods for starting the EMR 760. Either by direct activation or by setting the XST and ON bits of control register 1. There is really no choice. The XST has to be used. This provides the time code converters the necessary control over the activation of the EMR 760. All registers, including

control register 1, will be initialized by the computer. The time code converters will detect the beginning of the data stream. The time code converters will immediately clear the XST bit and the EMR 760 will be activated. The selection of the external start was the final selection of operation modes on the EMR 760. To summarize the selections chosen the following is a list of the modes utilized on the EMR 760:

- * Cyclic mode with two buffers each having the capacity of 16,384 sixteen bit words.
- * XST will be used in conjunction with the ON bit in control register 1. It will be cleared by the time code converters to activate the EMR 760.
- * Interrupt 2 will be used to stop the transfer session and will be controlled by the time code converters.
- * Bursting will not be used for this simple application. It should be considered in subsequent applications.

Generally, there has been discussion on both the telemetry equipment, the input equipment, and the EMR 760, the equipment used as a medium for transfer. Very little discussion has progressed on the VAX-11/780. This has been intentional because the best should always be saved for last (well, not quite last).

VAX-11/780

The VAX-11/780 is the Virtual Address Extension of the PDP-11 family of computers. DEC provides a very sophisticated and very, very complex virtual memory operating system (VAX/VMS) to support the VAX-11. Any information on the hardware, software, or architecture of the VAX-11 that will be discussed in this section can be obtained in either the Hardware (Ref 2), Software (Ref 1), or Architecture (Ref 3) Handbooks provided by DEC. The purpose of this discussion of the VAX-11 computer is not to explain the system and associated software. Rather, this section will provide the reader with an understanding of the system in relation to external devices, specifically the EMR 760. This section will take the reader from the very basic level of the operating system, that of the user environment, to the most privileged level of the operating system, that of the kernal environment. This section will also discuss, briefly, the nature of synchronization in the scheduler of the VAX-11. It is assumed that the reader has had prior knowledge of operating systems in general. Finally, this section will discuss, in some detail, the UNIBUS subsystem of the VAX-11. Most of this section will be devoted to the UNIBUS subsystem because that is the natural environment of the EMR 760. Only that portion of the VAX-11 dedicated to external devices and their controllers will be discussed. For more de-

tailed information on the VAX-11, the reader is, again, referred to the three handbooks mentioned previously.

Environments. The VAX-11/780 is a powerful computer that falls into two categories: minicomputers and mainframes. The VAX-11/780 can be considered a mainframe because it provides the user with all the conveniences a mainframe computer has: a fast time-sharing environment, an effectively unlimited amount of file space, a vast number of powerful peripherals (high-speed disks, magnetic tape drives, operator console, and any number of terminals), a very large file execution environment (4 gigabytes), and the capability for network connection. The VAX-11/780 is also considered a minicomputer because of its physical size, ability to handle time-critical applications, expandibility with user supplied devices, and very simply because it is just a powerful extension of the PDP-11. To maintain these multitudes of environments, the VAX/VMS provides four operating modes: kernel (most privileged), executive, supervisor, and user (least privileged).

The user mode (or user environment) of the system provides the user with the minimum of privileges needed to perform user related functions: execute tasks that do not have any affect on the system or its structures. The user can access disk files, read data from a terminal, and perform other menial tasks. The user is not given any privileges that would allow access to portions of the operating system

and related data structures. The supervisor mode is the next to least privileged environment. It is used to execute operating system functions such as command interpretation (Ref 2: 107). The next mode on the "access mode ladder" is the executive. The executive mode is the second most privileged mode (or environment). The executive mode is used when executing system service routines; such as system timer routines, system routines for disk manipulation, and the record management routines. The kernel (or most privileged) mode is used by the operating system for system page management, I/O drivers, and scheduling. Given adequate privileges, any user can enter these access modes. Scheduling is performed on a priority basis.

The scheduler assigns a base priority of zero to all jobs when it initiates execution. It is the jobs responsibility to raise that priority to the desired value. The priority is a number between 0 and 31. Zero is considered the lowest priority and all user programs start out at this level. To raise that priority level the user must have the privilege to alter priorities. These priorities are generally the only priorities the user will ever need. For time critical environments, programs can adjust their interrupt priority level. The processor recognizes 32 priority levels (0-31). The upper 16 priorities are allocated to interrupts generated by hardware. While, the lower 16 priorities are reserved for software interrupts. If an interrupt is requested that is greater than the current priority for the

process in execution, that process is preempted and the one with the higher priority takes control of the system. This allows a time-critical application to take complete control of the system while it performs its task.

The VAX/VMS provides the user with a virtual addressing space of 4 gigabytes. VAX/VMS accomplishes this through a page management system. Any addressing the user performs is on this virtual address space. VAX/VMS page management system continually updates virtual page tables and system page tables (Figure 15). As illustrated in Figure 15, every virtual addresses has a system virtual address page table entry hidden in its bit structure. The SVAPTE's point to a location in the virtual address page table. Every entry in the virtual address page table points to an entry in the system address page table. This entry then points to a page in physical memory. The byte offset from the original virtual address is carried through the entire operation and is used with the system page table entry to locate a particular physical, byte-addressable location. For an example of this translation the user is referred to ref 2: Appendix F. The system page table uses 30 out of the 32 bits in an address to locate an address in memory. The other two bits are controls to distinguish a read from a write.

The architecture of the VAX-11/780 is depicted in Figure 16. The VAX-11 uses three bus structures: an SBI, a UNIBUS, and a MASSBUS. The SBI (Synchronous Backplane In-

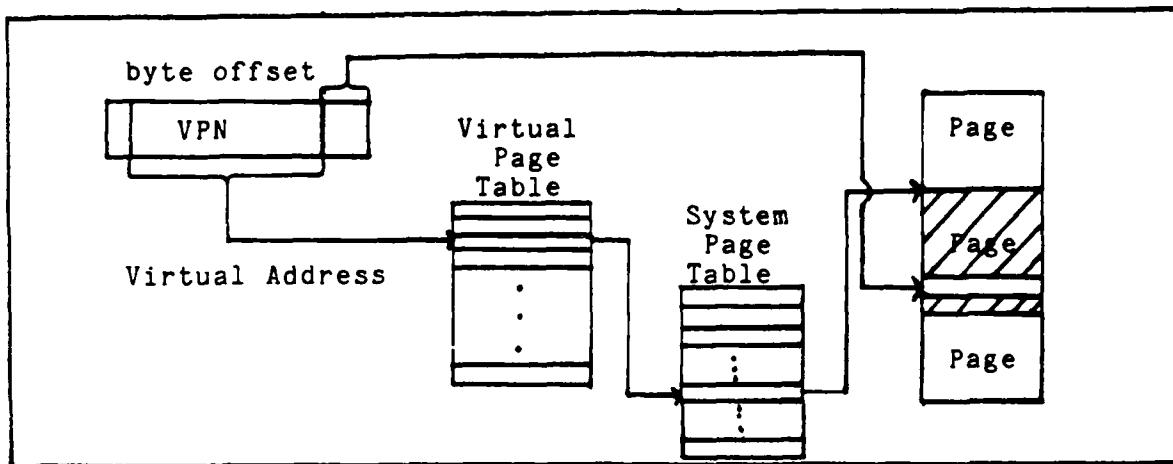


Figure 15. Page Tables

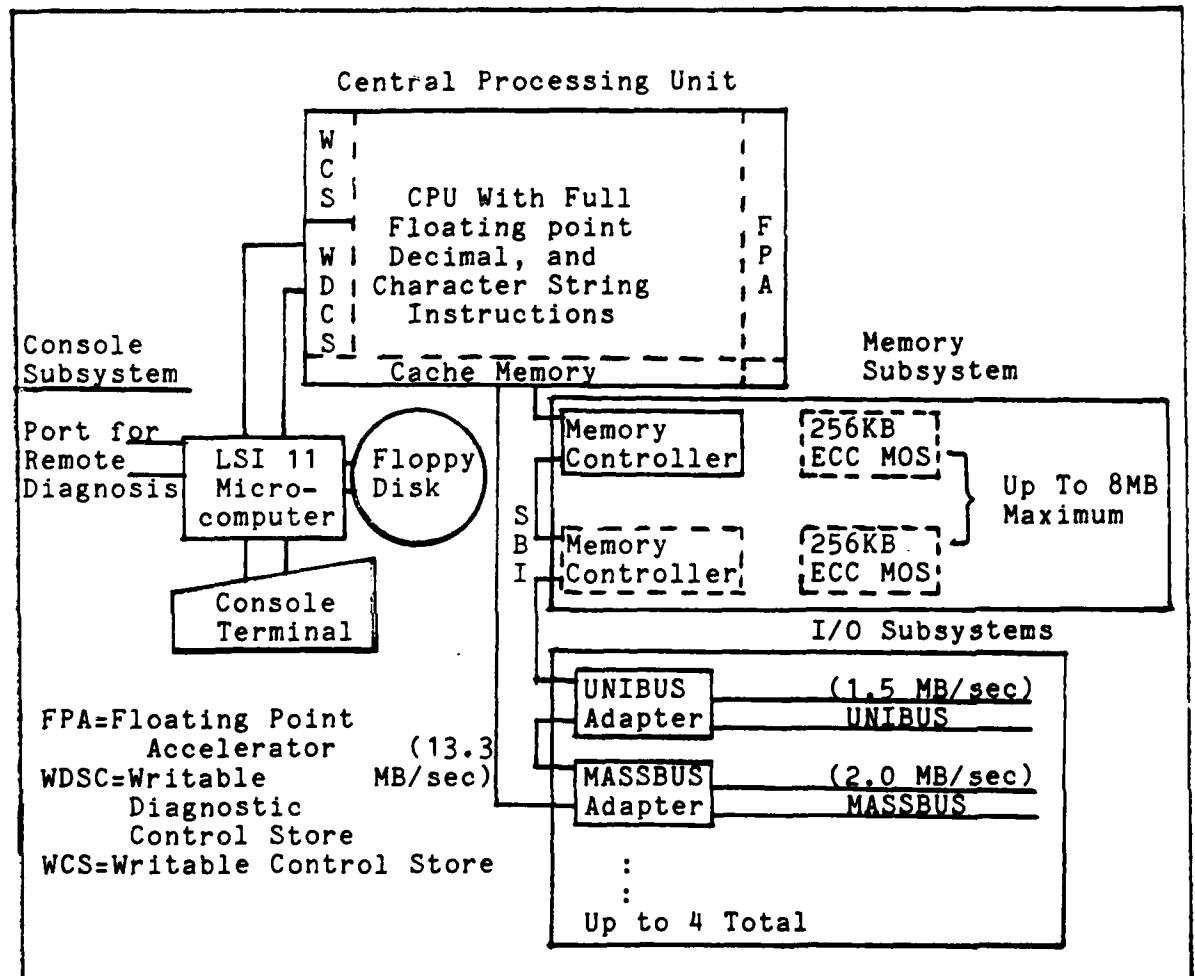


Figure 16. VAX-11/780 Hardware Configuration
(Ref 3: Figure 1-1)

terconnect) is used as the interface between the CPU, memory, UNIBUS, and the MASSBUS. The bus is allocated on a "master-slave" relationship. Everything except the memory can declare itself the master and can send up to 64-bits of information to the slave. All synchronization is performed in a handshake environment so neither the master nor the slave need to give full control to the other. The actual interface between the SBI environment and either the MASSBUS or UNIBUS environment is handled by adapters. These adapters perform all translations necessary for information to flow unobstructed from one environment to the other. Together, the UNIBUS and UNIBUS adapter are considered the UNIBUS subsystem.

UNIBUS Subsystem. The UNIBUS subsystem contains a UNIBUS and a UNIBUS adapter. The UNIBUS is used in exactly the same context as the UNIBUS from the PDP-11. This is what provides the major compatibility between the PDP-11 and the VAX-11. The UNIBUS adapter provides the mapping needed by the UNIBUS to allow data to transfer across the SBI bus. There are several aspects of the UNIBUS and its adapter that deserve merit at this time. First the adapter registers that are used in mapping a UNIBUS address to an SBI address. Second, the response the SBI gives to interrupts on the UNIBUS. Third, the data paths used to transfer data from the UNIBUS adapter to the SBI. Fourth, the transfer request numbers associated with the arbitration of the SBI bus.

The UNIBUS map registers are a very critical por-

tion of the analysis. Since the EMR 760 will be transferring data to SBI memory, it would be very helpful to have a clear understanding as to how the map registers will be used and what is needed to initialize them. There are two possible operations the map registers are involved in, the translation from the UNIBUS to the SBI and the translation from the SBI to the UNIBUS (this is needed to access I/O device registers)(Ref 2:175-178). Figure 17 illustrates the translation that is needed to transfer an 18-bit UNIBUS address to a 30-bit SBI address. The upper nine bits of the UNIBUS address contain a map register number (Figure 17). This map register number is an index into the UNIBUS adapter registers. The map register in the UNIBUS adapter will contain a 21-bit page frame number (Figure 17). This page frame number is used as a physical page index into the SBI memory. Bits 2-8 of the UNIBUS address contain the longword offset into the page indicated by the page frame number. Bits 0-1 and the two control bits are used to specify what function is to be performed. This is accomplished by a function mask encode (Figure 17). The interpretation of the function field specifiers is illustrated in Table I. The only need for Table I is to illustrate that there is an encoder and that there will be an interpretable mask from it. To recap, the SBI address contains a page frame number that was obtained from a map register which was indexed by a map register number in the UNIBUS address. The SBI address also contains a longword that was taken directly from the UNIBUS

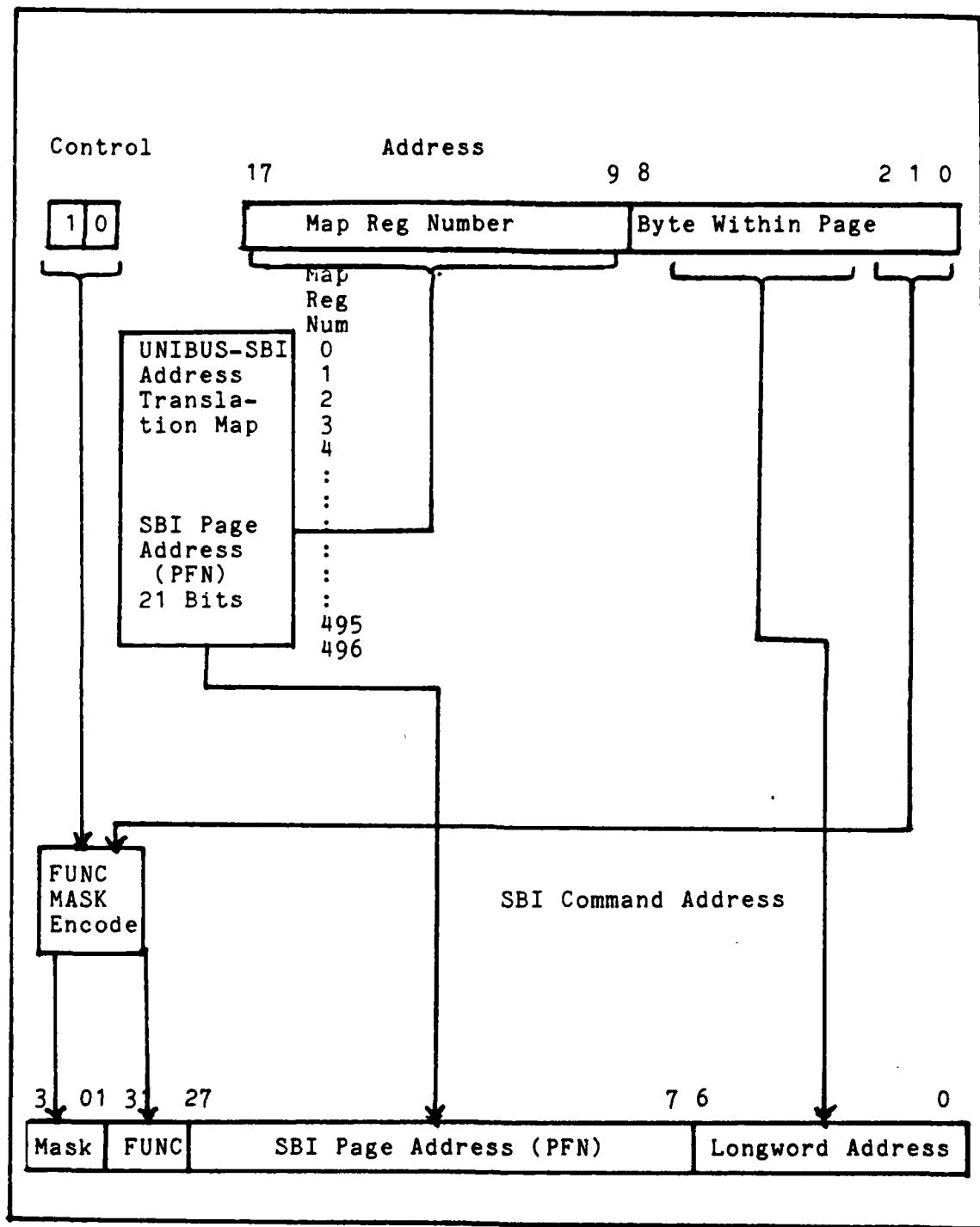


Figure 17. UNIBUS to SBI Address Translation
(Ref 2: Figure 9-5)

address. Finally, bits 0-1 are used in conjunction with two control bits that mask into an SBI function. Well, what about transferring the other way? Namely, from an SBI address to a UNIBUS address.

It seems that transferring from an SBI address to a UNIBUS address shouldn't be too difficult because there are 30 bits in an SBI address and only 18 bits in a UNIBUS address (Ref 2:170-172). This is a fair assumption and as will be seen the translation is not as complicated as the one just described. Table I represents the translation needed for an SBI address to be converted to a UNIBUS address. The mask and function bits of the SBI address get converted in the same fashion as before (Figure 14, Table II). They are sent through the same type of encoder and provide the control bits and the word addressed in the UNIBUS (Figure 18). Table II is similar to Table I and is used in the same context. Bits 0-15 transfer directly to the UNIBUS address while bits 16-17 indicate which UNIBUS address space the UNIBUS is residing in (Figure 18). The UNIBUS can reside in one of four places (Figure 18). Their UNIBUS addresses corresponding to their physical addresses are shown in Figure 18. So far the transfer from SBI to UNIBUS and the transfer from UNIBUS to SBI have been demonstrated, but why were only 21 of 32 bits used in the UNIBUS adapter map registers?

Actually, all but bits 27-30 of the map registers

are used (Figure 19)(Ref 2:213-216). Bit 31 is a map register valid bit (Figure 19). This bit is used when a process initializes the map register. It will also save the UNIBUS adapter from trying to access an address that wasn't really supposed to be used. When the map registers are initially loaded the final map register is given a not-valid assignment and if the UNIBUS adapter ever tries to access that map register, an error will be generated. Another interesting field in the map registers is the data path designator (Figure 19).

The data path designator field of the UNIBUS map registers, refers to the number of the data path used to transfer the incoming data from the UNIBUS adapter to the SBI bus. There are 16 data paths available on the UNIBUS adapter. Data path 0 is the direct data path and transfers data directly to the SBI bus. In contrast, the other 15 data paths are considered buffered data paths and can transfer 16-64 bits from the UNIBUS adapter to the SBI bus. Obviously, using the buffered data paths will save a considerable amount of time and will free the SBI bus for other operations. These data paths are software selectable and are initialized by the data path designator in the UNIBUS adapter map registers.

Another important consideration to the application that will be used by the EMR 760 is the handling of interrupts by the processor to devices on the UNIBUS. The UNIBUS

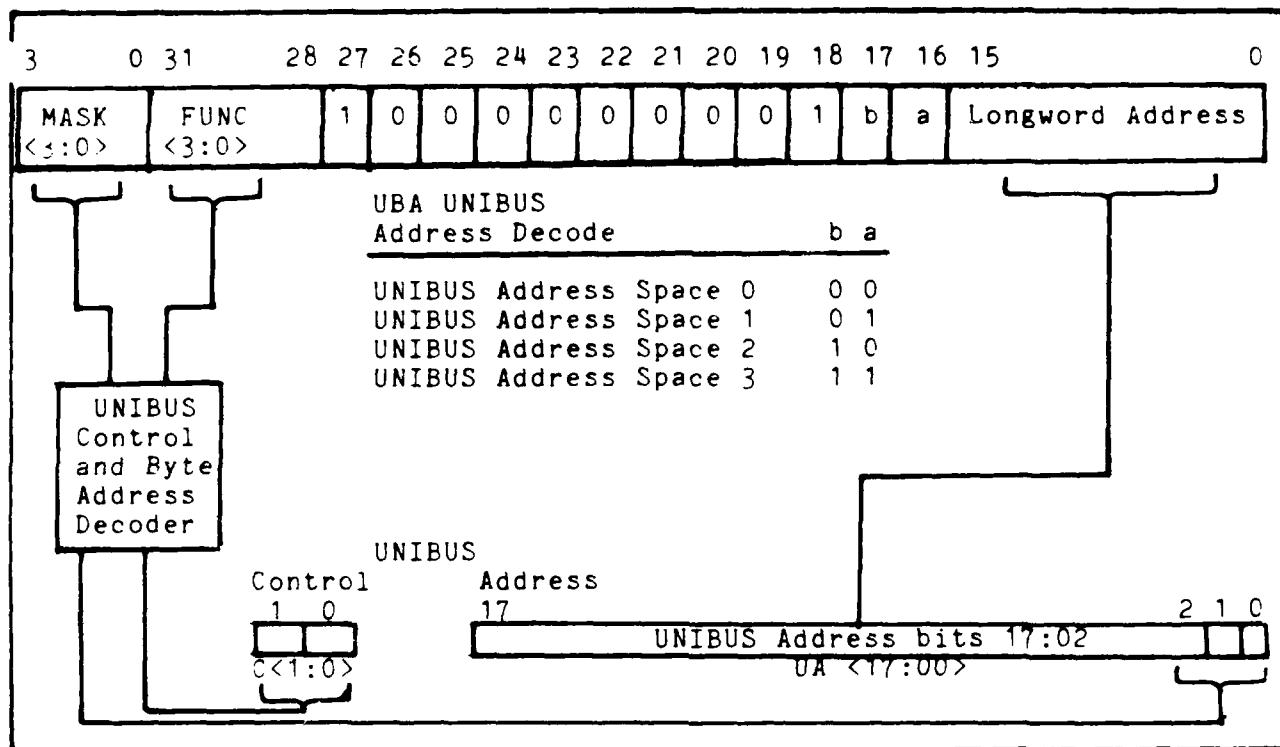


Figure 18. SBI To UNIBUS Control Address Translation
(Ref 2: Figure 9-4)

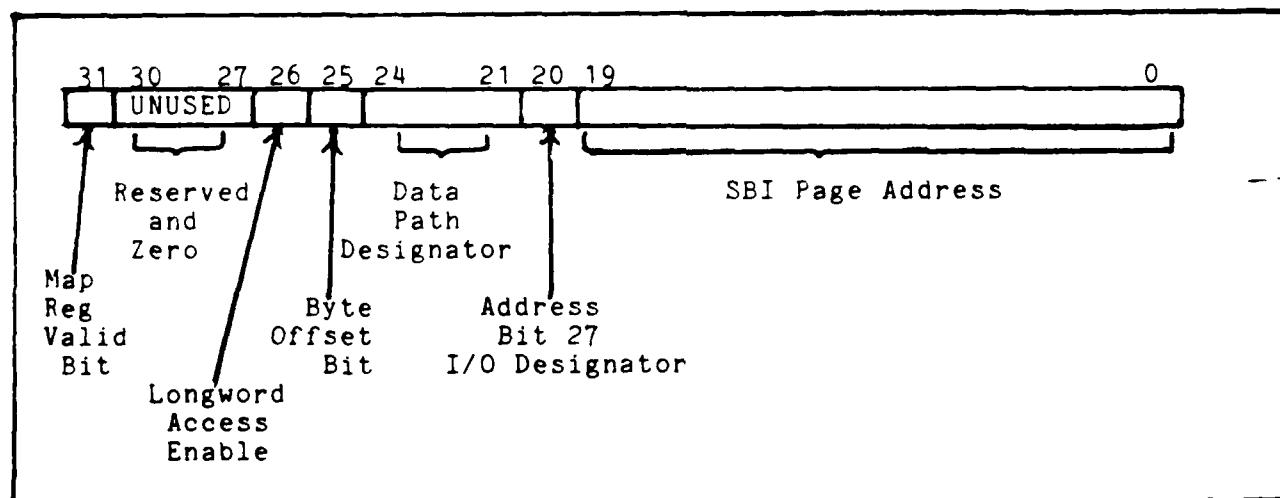


Figure 19. Map register Bit Configuration
(Ref 2: Figure 9-19)

maintains a level of priority for devices through the UNIBUS adapter. A device, connected to the UNIBUS, can generate one of five priority levels on the UNIBUS: NPR (Non-Processor Request), and BR4-7 (Bus Request levels 4, lowest priority, through 7, highest priority). The NPR maintains the highest priority because it will not use the processor during its transfer operation (it will perform a DMA operation). A chain effect is another priority mechanism on the UNIBUS. Chaining is just one device receiving a bus grant signal and if it had generated an interrupt, it will not transmit the signal on to devices on the same request level. Obviously, there would be a separate chain for each request level or the devices chained in a priority fashion.

When a device requests an interrupt on the UNIBUS, the UNIBUS adapter receives it. Through the use of an adapter control block the UNIBUS adapter finds the corresponding processor priority level associated with the device that requested the interrupt. This priority level had been initialized during setup and as stated in a previous section of this chapter, was between 16 and 31. When the UNIBUS adapter locates this interrupt priority level, it requests an interrupt on the SBI with that interrupt level. If this interrupt priority level is greater than the priority level of the current job executing, the processor will grant the interrupt and control will be transferred to the interrupt service routine for the device that generated the interrupt.

Table I. UNIBUS Field to SBI Field Translation
(Ref 2: Table 1-6)

UNIBUS		SBI	
Control C<1:0>	Byte Address	Function FUNC<3:0>	Mask M<3:0>
	A<1:0>		3 2 1 0
DATI	0 0	READ Mask	0 0 1 1
	1 0		1 1 0 0
DATO	0 0	WRITE MASK	0 0 1 1
	1 0		1 1 0 0
DATOB	0 0	WRITE MASK	0 0 0 1
	0 1		0 0 1 0
	1 0		0 1 0 0
	1 1		1 0 0 0
DATIP	0 0	INTERLOCK READ MASK	0 0 1 1
	1 0		1 1 0 0
followed by		INTERLOCK WRITE MASK	
DATO	0 0		0 0 1 1
	1 0		1 1 0 0
or		INTERLOCK WRITE MASK	
DATOB	0 0		0 0 0 1
	0 1		0 0 1 0
	1 0		0 1 0 0
	0 1		1 0 0 0

Table III. UNIBUS Device Address Space
(Ref 2: Figure 9-2)

UNIBUS I/O Address Space	UNIBUS Address (Octal)	Physical Byte Locations(Hex)
UNIBUS 0 Address Space(tr-value 3)	760000-777777	2013E000-2013FFFF
UNIBUS 1 Address Space(tr-value 4)	760000-777777	2017E000-2017FFFF
UNIBUS 2 Address Space(tr-value 5)	760000-777777	201BE000-201BFFFF
UNIBUS 3 Address Space(tr-value 6)	760000-777777	201FE000-201FFFFF

Table II. SBI Function-Mask Translation To UNIBUS Control-Address
 (Ref 2: Table 9-4)

SBI		UNIBUS	
Function <3:0>	MASK 3 2 1 0	Control C<1:0>	Address UA<1:0>
Read Mask	0 0 0 1	DATI	0 0
	0 0 1 1	DATI	0 0
	0 0 1 0	DATI	0 0
	0 1 0 0	DATI	1 0
	1 1 0 0	DATI	1 0
	1 0 0 0	DATI	1 0
Write Mask	0 0 0 1	DATOB	0 0
	0 0 1 0	DATOB	0 1
	0 1 0 0	DATOB	1 0
	1 0 0 0	DATOB	1 1
	0 0 1 1	DATO	0 0
	1 1 0 0	DATO	1 0
Interlock Read Mask (Sets Interlock Flip Flop For DATIP- DATO Sequence)	0 0 0 1	DATIP	0 0
	0 0 1 0	DATIP	0 0
	0 1 0 0	DATIP	1 0
	1 0 0 0	DATIP	1 0
	0 0 1 1	DATIP	0 0
	1 1 0 0	DATIP	1 0
Interlock Write Mask	0 0 0 1	DATOB	0 0
	0 0 1 0	DATOB	0 1
	0 1 0 0	DATOB	1 0
	1 0 0 0	DATOB	1 1
	0 0 1 1	DATO	0 0
	1 1 0 0	DATO	1 0

The UNIBUS adaptor will locate the address of the interrupt service routine through a series of pointers that again originate in the adapter control block.

The UNIBUS adapter also has a priority constraint associated with it. This is called the SBI arbitration line. It is also referred to as the tr-value. The transfer request value (tr-value) and the corresponding subsystem assignments are illustrated in Table III. As can be seen the UNIBUS has a lesser tr-value than any of the MASSBUS adapters. If two requests for the use of the SBI are generated at the same time, the request from the higher arbitration (tr-value) line will be granted. So, there are a great number of mechanisms the VAX-11 uses (hardware and software) to synchronize the execution of multiple bus structures.

This concludes the introduction of the VAX-11/780 and if the reader is confused (and he very well should be) he should read the handbooks provided by DEC. With the environment of the VAX-11 defined, the next step would be to select the different aspects of that environment that would be beneficial to the EMR 760. Specifically, what UNIBUS paths, lines and requests the EMR 760 will use.

Selection of Priorities A lot of effort has been devoted to explaining the priority structure of the VAX-11 processor and its UNIBUS. The question now is how will the EMR 760 utilize these priorities to its advantage? Each section of priority assignment will be discussed and a selection

made.

Environment. The environment is restricted to that of the kernel, or most privileged, mode. This restriction will be placed on the program that drives the EMR 760 because it will, obviously, have to use the I/O space to access the device registers. Actually, the operating system takes care of the environment of the driver and it should be noted that the software interrupt priority level of the driver will be 8. The hardware interrupt priority level for the device will be 23. The reason for these interrupt priority levels will be discussed in the next chapter. The drivers are actually loaded into the operating system and are treated as operating routines.

UNIBUS The modes of the EMR 760 were discussed earlier and the reader was given just a glimpse of the requirements affecting the UNIBUS subsystem. Other requirements include: the placement of the EMR 760 on the UNIBUS, what UNIBUS the EMR is placed on, the data path to be used in the data transfer, and the bus request level to be used by the EMR 760.

The bus request level for a hardware interrupt can range from 4 through 7. Since the EMR 760 is a very time critical device, a bus request level of 7 will be used. This does not affect the NPR request which will be generated every time a data transfer occurs. ..ough, the physical placement of the UNIBUS will affect the NPR request. If

there is more than one UNIBUS in the system, the EMR 760 should be placed on the UNIBUS with the greater SBI arbitration line. This will give the EMR 760 a slight edge over the subsequent UNIBUS's. Along the same line, if there is more than one device on the EMR 760's UNIBUS then the EMR 760 should be put at the beginning of the chain. This will also provide a slight advantage over other devices connected to the same UNIBUS. If all these restrictions are adhered to, the EMR 760 will become the most time-critical device on all UNIBUS subsystems and will take precedence over every other device.

Summary

To summarize, this chapter has been devoted to discussing the system requirements and constraints pertaining to the three major hardware systems: the telemetry equipment, the EMR 760, and the VAX-11/780.

The telemetry equipment was found to constrain the EMR 760 to only a limited number of operating modes (cyclic, two buffer, external start). Other operating modes were discussed in detail and a selection was made based upon the overall objective of this research. The EMR 760 was further constrained by the operating environment of the VAX-11/780. The VAX-11/780 only provided a limited number of buffer areas and the operating system was shown to have a consider-

able effect on any software written to control or drive the EMR 760. The following chapter provides, in greater detail, the software necessary to control or drive the EMR 760. It will provide a discussion of the routines and programming constraints VAX/VMS places on all device drivers.

III. Development of a Device Driver

The purpose of this chapter is to define the tables and routines of a typical device driver. This chapter will also discuss the methods used by the operating system to control the device driver. This chapter will progress from the definition of routines normally found in a device driver to the synchronization used by VAX/VMS and will end with a description of the I/O data base.

Device Driver

A driver for a device on the VAX-11/780 must have tables and routines to define the device to the operating system and handle all I/O requests on the device. A driver will contain, if necessary, the following items (Ref 5:1-1,1-2):

- * Preprocessing routines. Preprocessing routines will be used to validate user supplied parameters specified in the Queue I/O system service, format data, and lock user supplied buffer areas in memory. These preprocessing routines are commonly referred to as function decision table (FDT) routines.
- * Start I/O routine. The start I/O routine will load

device registers, request map registers and data paths, and return a completion status to the user.

- * Interrupt Service routine. The interrupt routine will service any hardware interrupts requested from the device. The interrupt routine will store device registers temporarily in the I/O data base and reactivate the start I/O routine. The start I/O routine will use the information in the I/O data base to complete the I/O operation.
- * Error Recovery routine. The error recovery routine will attempt retry of an operation that failed. This will depend on the device and the application needed by the user. The EMR 760 driver does not use an error recovery routine.
- * Error Logging routine. An error logging routine is used to dump device registers to a user supplied buffer when an error is detected. This feature is not included in the EMR 760 driver.
- * Cancel I/O routine. The cancel I/O routine will terminate any I/O operations still active on the device. VAX/VMS provides a device independent routine to perform this function and it will be used in the EMR 760 driver.
- * Initialization routine. The initialization routines ready a device and controller for subsequent

I/O operations.

- * Driver Prologue Table. The driver prologue table describes the driver to the operating system. It provides the characteristics of the device and driver to VAX/VMS. It also supplies initialization of fields in the I/O data base when power is turned on and when the driver is reloaded into the system.
- * Driver Dispatch Table. This table defines the entry addresses to the driver's routines.
- * Function Decision Table. This table defines the functions that can be performed on the device and the entry addresses to routines that will be used to preprocess the I/O request.

VAX/VMS is in charge of synchronizing the execution of all drivers on the system. VAX/VMS uses fork processes, interrupt priority levels, fork queues, and resource wait queues to synchronize the execution of all drivers (Ref 5:1-3).

Fork Process

A fork process is a synchronization mechanism used by the operating system to define the context of most driver routines. A fork process has minimal context and allows the

processor to switch between fork processes with very little overhead. The fork process is defined by (Ref 5:1-3):

- * Three general registers.
- * A program counter.
- * A unit control block (UCB). This is part of the I/O data base and will be described later in this chapter.

Fork processes reside in the system space and can not access the virtual portion of the machine. Fork processes execute at a raised interrupt priority level and request software interrupts to start execution.

Interrupt Priority Level

Interrupt priority levels (IPL) are used as a synchronization mechanism by the operating system. Every process has an interrupt priority level. This priority level will allow the preemption of executing tasks if the priority level is lower than the one requested. A higher priority level will always take precedence over lower priority levels. A user process executes at an IPL of 0 and will be preempted by any routine with a nonzero priority level. The following IPLs are used to define the levels used in device drivers (Ref 5:1-7,1-8):

- * Hardware device IPLs. The hardware device IPL will be used to service a hardware interrupt requested by the device. This can be a number from 20 through 23 and is specified in the driver prologue table.
- * Driver Fork IPLs. The driver fork IPL will be used to schedule execution of the driver fork process. This can range from 8 through 11 and will be specified in the driver prologue table.
- * I/O Completion IPL. VAX/VMS needs to perform steps to complete the I/O operation and the I/O completion IPL will be used to schedule the execution of these steps. Since it is more important to begin an I/O request, the completion IPL has a value of 4 and is set by the operating system.
- * AST Delivery IPL. An AST is an Asynchronous System Trap routine that can be specified by the user in the Queue I/O system service. The AST routine is executed as the final step in the completion of the I/O request. It has an IPL of 2 and is set by the operating system.

The IPL is used to synchronize the execution of driver routines. The fork IPL is used when a request from the fork process is interpreted by the operating system. While the

fork process is awaiting execution, it is placed in a fork queue.

Fork Queue

A fork queue is used to establish a method for executing several fork processes of the same IPL. The only information needed by the fork queue is the address of the UCB associated with the fork process. The UCB contains all information needed to start the fork process. Each fork IPL has a separate fork queue (Ref 5:1-8). The fork queue is used to schedule fork processes awaiting execution. Another queue used to schedule fork processes is the resource wait queue.

Resource Wait Queue

The resource wait queue is identical to a fork process queue except that it schedules fork processes that have been suspended because of resources not available. Throughout the execution of the driver, several resources will be needed (Ref 5:1-8):

- * Central Processor. The central processor is the resource needed by all drivers and has to be shared among all fork processes executing in the system. Fork processes waiting for this resource will use

the fork queue.

- * UNIBUS Map Registers. If the driver will be transferring data over the UNIBUS, map registers will need to be allocated.
- * UNIBUS Data Paths. If the driver will be transferring data over the UNIBUS, a data path will need to be allocated.
- * Controller Data Channel. If the driver controls several devices, a controller data channel will need to be requested.

If the driver requests resources that are not available, the operating system will suspend the fork process until another fork process has released the resources. The resource wait queue uses the addresses of the devices UCB when it can fulfill a resource request. The UCB is part of the I/O data base and is used by most routines in the driver.

I/O Data Base

The I/O data base contains information needed to describe all components of an I/O operation. There are three parts to the I/O data base (Ref 5:1-5):

- * Driver Tables. These include the driver dispatch

table, driver prologue table, and function decision table for every driver in the system. The function of these routines was discussed earlier.

- * Control Blocks. The control blocks describe the different components used in an I/O operation. There are control blocks to describe the bus adapters, types of devices, device units, controllers, the logical path (channel) from a process to a device.
- * I/O Request Packets. The request packets are used throughout the execution of the driver and they describe the individual I/O operations on a device.

The driver tables were discussed earlier. The control blocks consist of a DDB, UCB, CRB, IDB, ADP, and a CCB.

Control Blocks. The control blocks describe the components used in an I/O operation. There are six types of control blocks in the I/O data base (Ref 5:1-6,1-7):

- * Device Data Block (DDB). The DDB contains information on device types supported by a controller. There are separate DDBs for every device type. The DDB describes the generic name of the device, a controller designator, the driver name, and the UCB address of the first device type attached to the controller.

- * Unit Control Block (UCB). There is a UCB for every device in the system. The UCB describes the characteristics of the device, current state of the device, fork block, and a list head used by the fork queues. When a driver is interrupted, the UCB will hold the context of the driver and is used as the focal point of the I/O data base.
- * Channel Request Block (CRB). A channel (controller) request block is created for every controller on the system. The CRB describes the current state of the controller and a list of UCBs awaiting access to the controller. The CRB also contains code to dispatch interrupts to the routine specified for that driver.
- * Interrupt Data Block (IDB). The IDB further defines the CRB. The system creates an IDB for every controller. The IDB contains a list of device units associated with the controller, a pointer to the UCB being serviced, a pointer to the devices registers, and a pointer to the controllers adapter.
- * Adapter Control Block (ADP). An ADP defines the current state of a UNIBUS or MASSBUS adapter. An ADP is created for every adapter in the system. An ADP contains the queues and allocation bit maps

needed to control the resources of the adapter.

- * Channel Control Block (CCB). A channel is a logical path between a controller and the UCB of a device unit. When a process issues the Assign I/O channel system service, a description of the channel is written to the CCB.

The control blocks define the components used during an I/O operation. The I/O request packets define the individual I/O requests.

I/O Request Packets. The last part of the I/O data base is the I/O request packets. The I/O request packet describe individual I/O operations. It is used to hold device specific parameters specified in the Queue I/O system service (Ref 5:1-7). This includes a byte count, buffer address, function code, and a byte offset for a DMA read operation. The I/O request packet also contains pointers to a target device and other blocks in the I/O data base.

The I/O data base uses driver tables, control blocks and I/O request packets to describe I/O components and operations. To illustrate how each part of the I/O data base is used an overview of a DMA read operation will be presented.

Overview of DMA Read Operation

A DMA read operation will be used to illustrate the relationship of the I/O data base and the driver routines (Figure 20). There are 13 steps taken to complete the I/O operation (Ref 5:1-12,1-13):

1. A user process requests input from a device to buffer areas in its virtual address space. The process uses the Queue I/O system service to specify a target device, address of the buffer area, and a byte count.
2. The operating system will perform any necessary preprocessing. This includes validating the request and creating an I/O request packet. The I/O data base control blocks that describe the device and controller will be located. The operating system uses the function decision table to call a read function routine in the driver.
3. The driver performs I/O preprocessing in the function decision table (FDT) routine. The FDT routine checks the user accessibility to buffer areas and locks the buffer areas in memory. The read FDT routine will add any additional details to the I/O request packet and send the I/O request packet to

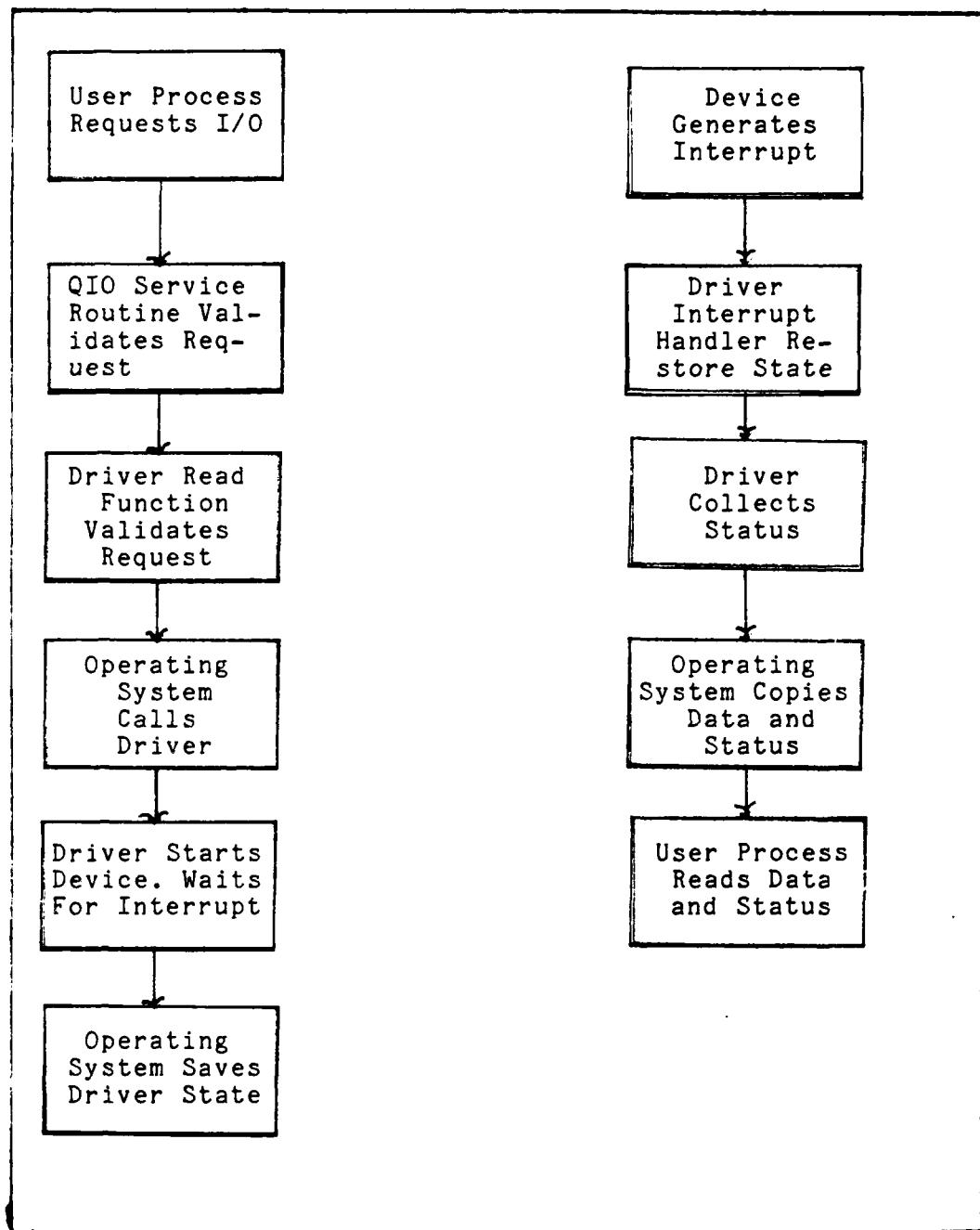


Figure 20. Processing a DMA Read Operation.
(Ref 5: Figure 1-3)

the driver.

4. VAX/VMS creates a fork process. VAX/VMS uses the fork block of the target UCB to insert the driver in a fork queue. VAX/VMS activates the driver fork process by calling the start I/O routine.
5. The driver start I/O routine readies the UNIBUS for the transfer. The start I/O routine uses the pointers in the UCB to interrogate the UNIBUS allocation bit map register. Map registers are allocated and subsequently loaded with information to convert the UNIBUS address to an SBI address.
6. The start I/O routine activates the device. The start I/O routine uses the pointers in the IDB to initialize device registers and activate the device.
7. The driver waits for an interrupt. After the driver has activated the device, the driver is suspended by saving its context in the UCB fork block. The fork block saves registers R3 and R4, the program counter, and the processor status longword. The processor is allowed to execute other processes until an interrupt or timeout has occurred.

8. The device requests an interrupt. When the device requests an interrupt, a general purpose interrupt routine in the UNIBUS adapter transfers control to the target CRB. The CRB activates the interrupt service routine for the device, via instructions in the CRB.
9. The driver services the interrupt. The driver interrupt service routine gains control with only a pointer to the IDB. Using this pointer, the interrupt service routine reads device registers and stores status of the operation in UCB fields. The interrupt routine will activate the suspended start I/O routine.
10. The operating system will insert the driver in a fork queue. The driver requests the remaining portion of the start I/O routine to complete as a fork process. Again, the fork block will be used to store the context of the driver. The driver will wait for activation by the fork dispatcher.
11. The fork dispatcher will activate the driver. When the fork IPL associated with the driver becomes the highest in the system, the fork dispatcher will activate the driver as a fork process.

12. The driver fork process completes device-dependant operations and returns status to the user via registers R0 and R1. The start I/O routine will purge the buffered data path, if used, and release map registers.
13. VAX/VMS completes the I/O operation. The VAX/VMS postprocessing routine will copy the I/O status into the user defined buffer area and will transfer control to the user process.

Summary

The device driver contains several routines and tables that are used to control the driver and access the I/O data base. The routines used in the EMR 760 driver include the initialization routines, start I/O routine, and two interrupt service routines, see appendix A for a listing of the driver.

VAX/VMS can switch between fork processes efficiently because every fork process has a fork block which contains the context of the fork process. The information in the fork block includes registers R3 and R4, the program counter, and the processor status longword.

The I/O data base contains information on I/O opera-

tions and I/O components. The I/O data base consists of driver tables, control blocks, and I/O request packets. The information in the I/O data base is used by the driver to complete an I/O operation.

The information described in this chapter addresses drivers in general with very little discussion on specifics of an EMR 760 driver. The next chapter will define the routines needed by an EMR 760 driver.

IV. Design of an EMR 760 Driver

The previous chapter discussed the tables and routines used by most drivers. This chapter will define the table and routines that are incorporated in the EMR 760 driver.

As was seen in the overview of I/O processing, a driver consists of a series of modular routines. The operating system provides the basis for transition between the driver routines. This has advantages as well as disadvantages. The main advantage is that the coupling and communication between the modules does not need to be elaborated. A disadvantage is the documentation of the system routines, especially when a device driver does not follow the standards. Without proper documentation, a minor action in one routine could affect the outcome of the entire driver.

The modular routines needed by a driver for an EMR 760 are:

- * Routines to initialize the device and controller.
- * Function Decision Table routine.
- * Start I/O routine. For initialization of the I/O process and completion of the I/O process.
- * Interrupt Service routines.

Controller Initialization Routine

The controller and device are initialized by a single initialization routine (Figure 21). The initialization routine defines the owner of the controller as the EMR 760 (or the UCB dedicated to the EMR 760), and, clears the two EMR 760 control registers, and, sets the online field in the device status word of the UCB. This concludes all initialization that is necessary. The initialization routine is called when the driver is connected to the system. The driver is called when a user program requests a read function on the EMR 760. The first routine that receives control is the Function Decision Table routine.

Function Decision Table Routine

A function decision table is used by the device driver to describe the permissible functions for that device. A driver for an EMR 760 has only one type of operation: read a block of data. Theoretically, any read block operation would be appropriate (READVBLK, READLBLK, READPBLK) but the READVBLK (read virtual block) function is the one used in a normal operation. The function decision table is used by the operating system to define the allowable functions for a device driver and the routines to execute in preparation for those functions. These routines translate virtual addresses to physical addresses and check the user accessibility to

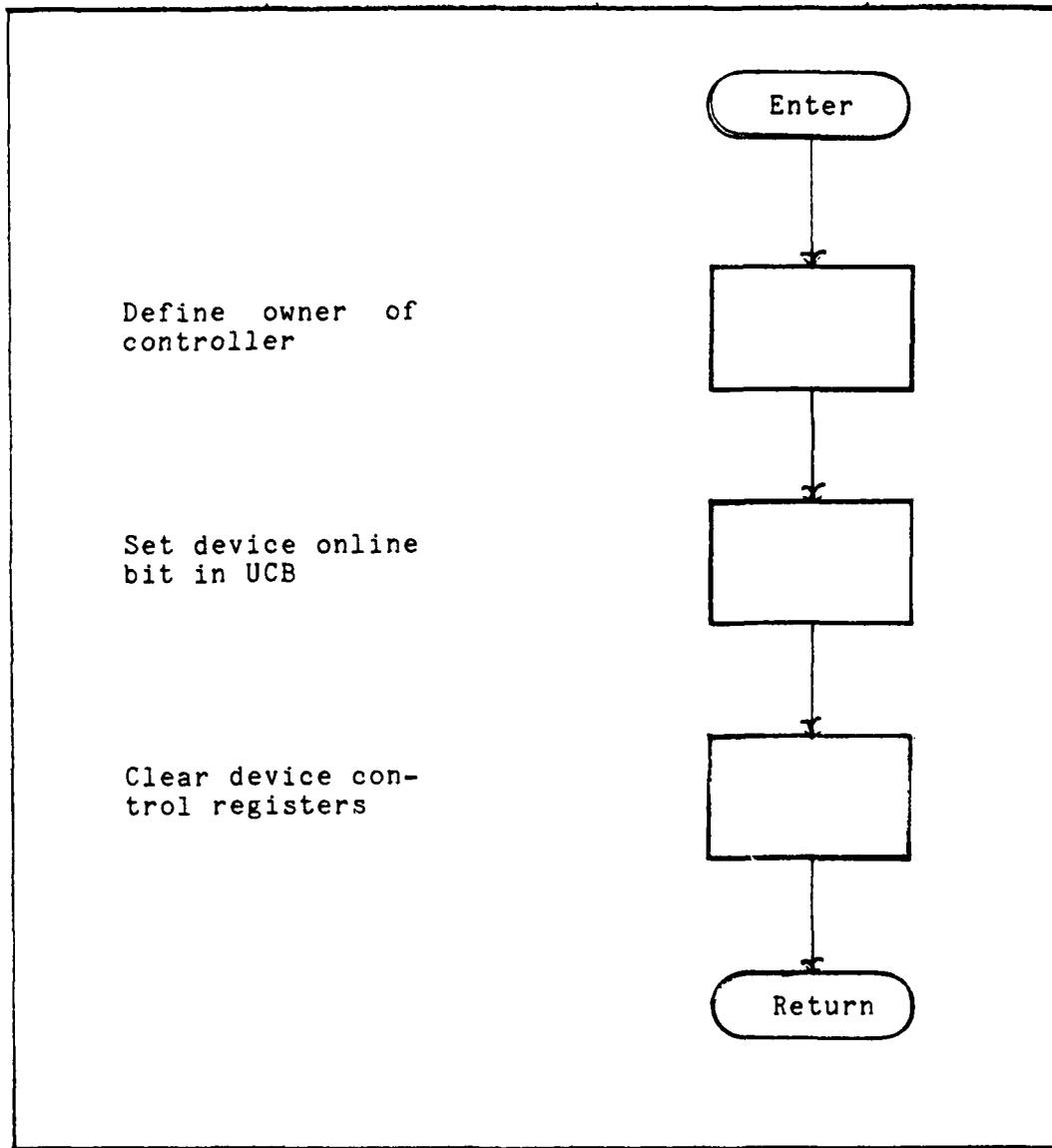


Figure 21. Initialization Routine

any buffer areas.

The function decision table routines play a very limited role in the process of an I/O operation with the EMR 760. DEC provides device independent routines for standard read and write operations. The EMR 760 uses the device independent routine EXE\$READ to perform the preprocessing needed for the EMR 760 driver. The flow of this program is illustrated in Figure 22. The FDT read routine checks the function code and write accessibility of the user. If these are satisfactory, the routine then locks the buffers in memory. The buffers need to be locked in memory because page faulting on a UNIBUS transfer will cause a system error. After performing all necessary functions, the FDT routine transfers control to a VAX/VMS packett queuing routine. This routine transfers the following IRP fields to their corresponding UCB fields:

- * IRP\$W_BCNT (Byte Count of the block transfer).
- * IRP\$W_BOFF (Byte Offset of starting buffer address).
- * IRP\$L_SVAPTE (System Virtual Address Page Table Entry of the starting address of the buffer area).

The packett queuing routine also initializes registers R3 and R5 with the address of the I/O request packett and the address of the UCB respectively. At this point the

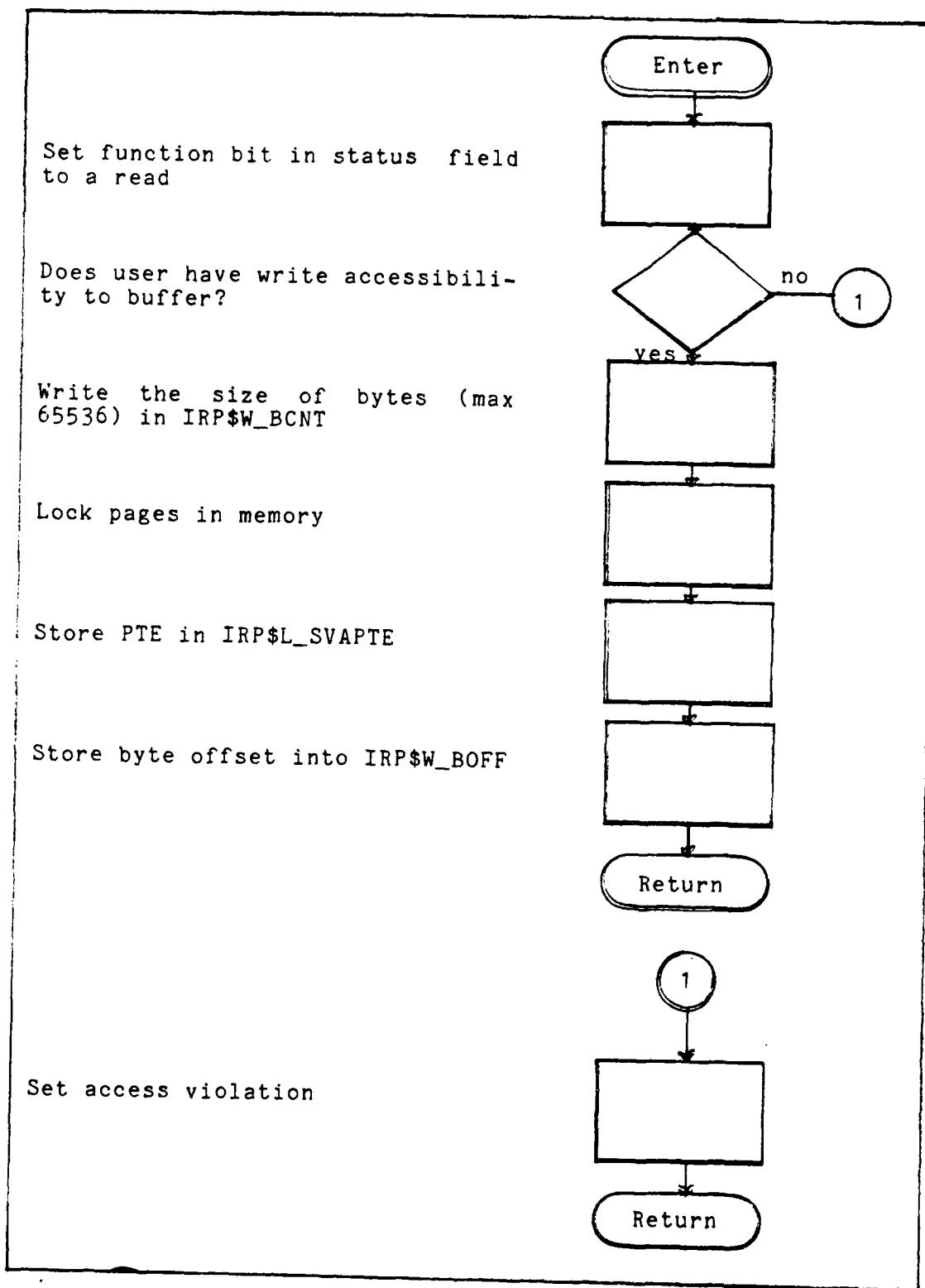


Figure 22. EXE\$READ Routine

queuing routine creates a fork process from the start I/O routine. It then sends the driver to a fork process queue to await execution. The start I/O routine is the only routine to execute as a fork process.

Start I/O Routine

The start I/O routine executes as a fork process. This is defined by the following restrictions (Ref 1:1-3):

- * System Mapping. Only system page tables are mapped. Therefore, the driver code cannot access virtual addresses in a process.
- * Kernel mode. Execution occurs in the most privileged access mode and can change IPL.
- * High IPL. The VAX/VMS routine that creates a driver fork process raises IPL to driver fork level before activating the driver. The driver can raise and lower the IPL between fork level and IPL\$POWER (31: the highest IPL).
- * Kernel or Interrupt Stack. Execution occurs on the kernel or interrupt stack. The driver must not alter the state of the stack without restoring it to its previous state before relinquishing control.

The start I/O routine is the workhorse of the EMR 760

driver. It is responsible for assigning all channels and UNIBUS map registers necessary for the transfer (Figure 23). This is made simple with the VAX/VMS routines that perform most of these functions. The start I/O routine clears the device status in the UCB and requests map registers. Requesting map registers consist of:

- * Interrogating the UNIBUS I/O register for map register allocation.
- * Allocating map registers. The number of map registers allocated is determined by the byte count and byte offset fields in the UCB. If there are enough consecutive map registers, the request is fulfilled. Otherwise the driver is suspended until the request can be fulfilled.

The start I/O routine will then load the map registers by calling a VAX/VMS routine. The load UNIBUS map register routine uses the UCB\$L_SVAPTE and a data path designator to load the UNIBUS map registers. The EMR 760 driver uses the direct data path (path 0). This is the default so there is no assignment necessary. After loading the map registers, the start I/O routine calculates the starting addresses of the two memory buffers. The SBI addresses are translated to UNIBUS addresses by several fields (Figure 17). These addresses are then stored in the device registers MA1 and MA2. Once the starting addresses have been transferred, the word

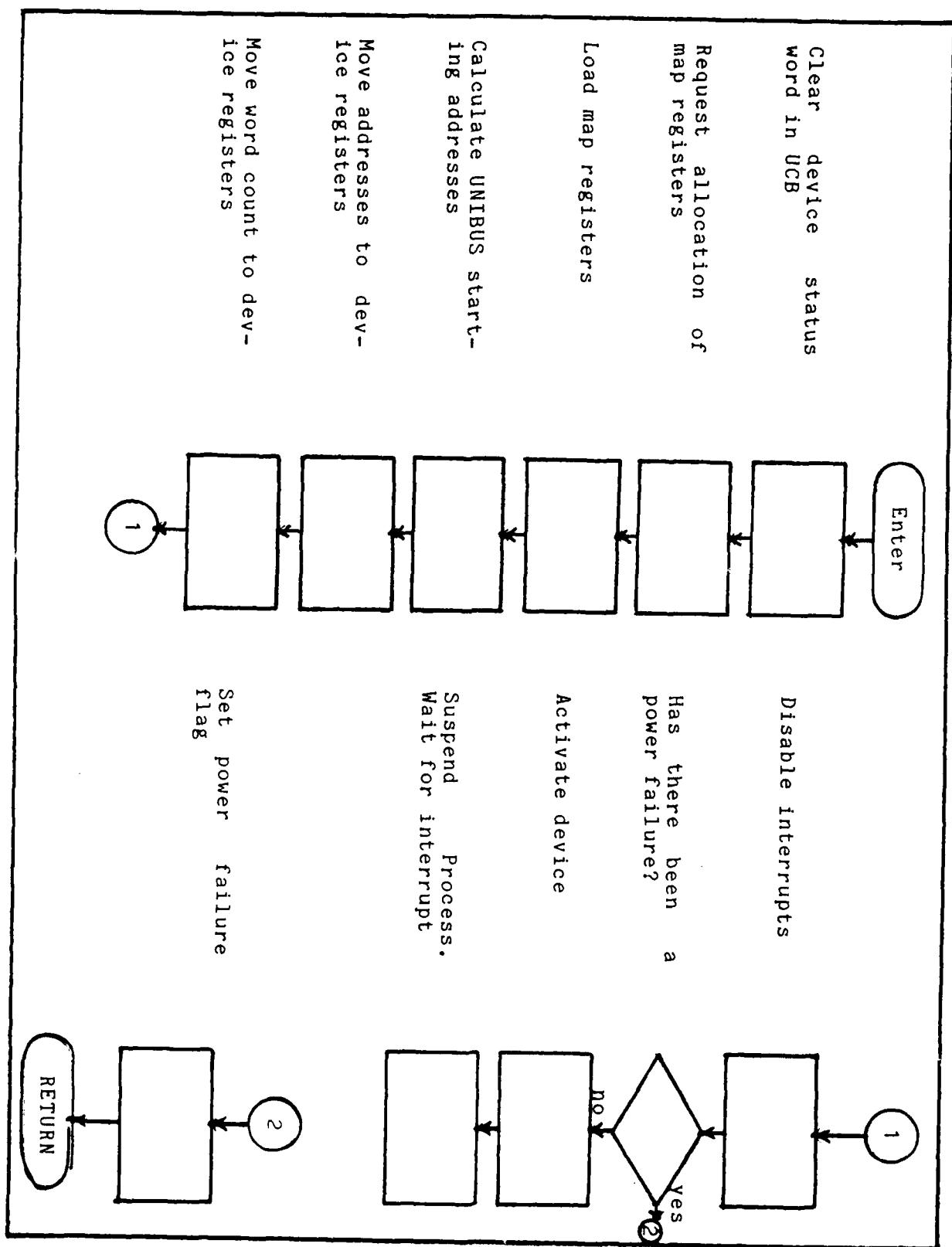


Figure 23. Start I/O Routine: Initialize
71

count is transferred to the WC1 and WC2 registers of the EMR 760. At this point, the EMR 760 can be activated, but, first the device status field needs to be interrogated to see if there has been a power failure since the operation was started. If the power has not failed, the FMR 760 is activated (by writing to control register 1). After the EMR 760 is activated, there is nothing more the start I/O routine can do until an interrupt occurs. So, the driver will suspend itself until either an interrupt is requested or a time out period is exceeded.

Now the driver is suspended by the wait for interrupt or timeout routine provided by VAX/VMS. This allows the user process that called the driver to continue execution. When an interrupt is finally requested, the only context of the driver is stored in the UCB fields. But, this is still a considerable amount of information:

- * A description of the I/O request and the state of the device.
- * The contents of R3 and R4.
- * The address of the UCB in the Interrupt Data Block (IDB).
- * A driver return address to the point just beyond the call for suspension.
- * The address of a driver timeout routine.

When a device requests an interrupt, VAX/VMS calls an interrupt service routine (ISR). The only information passed to the ISR is on the interrupt stack. This includes the address of the IDB, registers R0-R5, and the saved program counter and program status longword of the process executing before the interrupt was requested.

Interrupt Service Routine

For the EMR 760 application, there are two interrupt routines. One to handle a block end interrupt and one to handle an end of transfer interrupt. The only difference between the two routines is in the calling of the driver start I/O routine. The block end interrupt does not call the start I/O routine while the end of transfer does. This will allow the end of transfer interrupt to complete the I/O processing. The two routines are illustrated in Figures 24 and 25. Both routines clear the control registers of the EMR 760 and dismiss the interrupt with a return from interrupt instruction. The second interrupt routine activates the driver's start I/O routine where it was suspended, this allows postprocessing activities to begin.

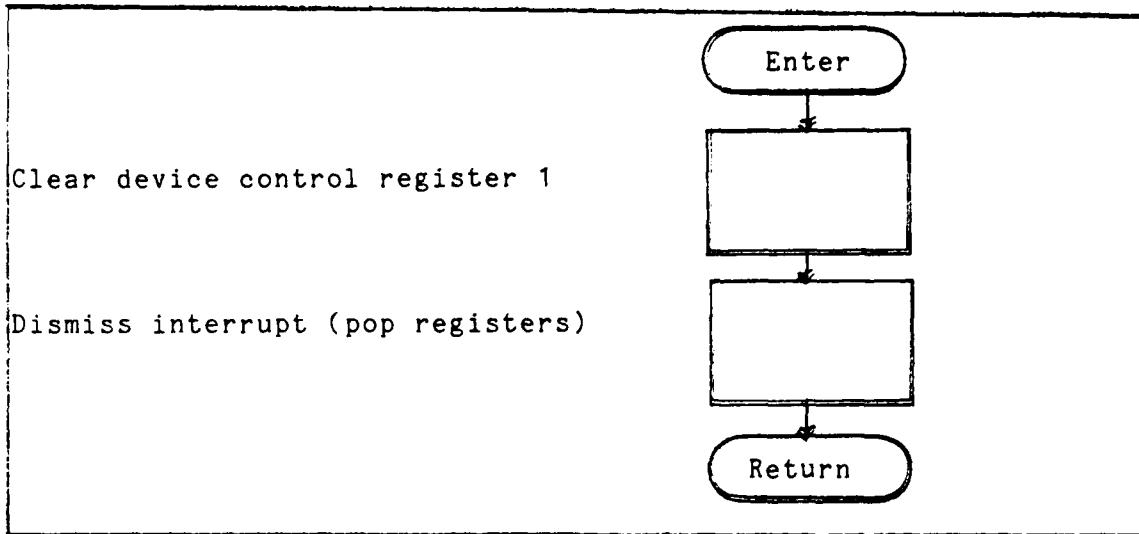


Figure 24. Interrupt Routine 1: Block End

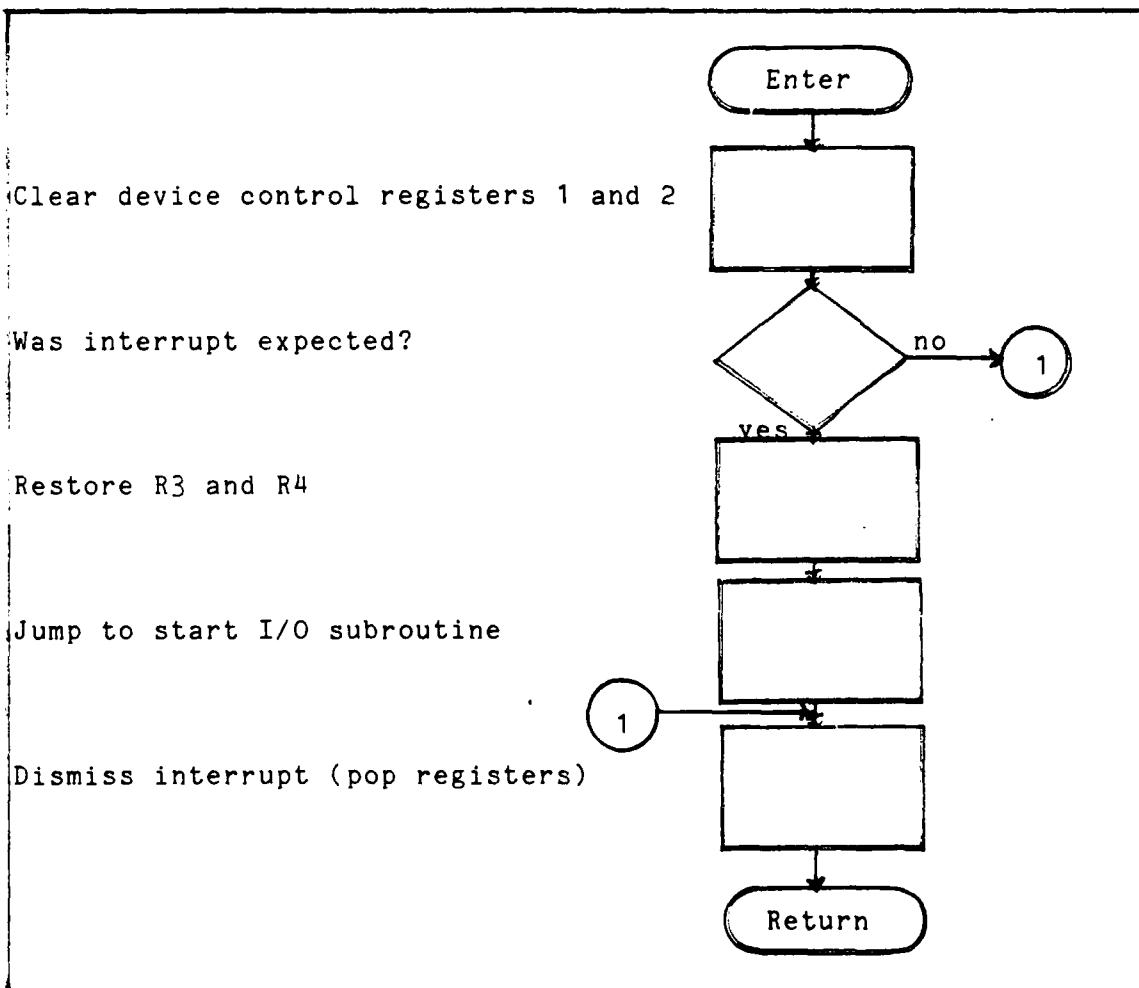


Figure 25. Interrupt Routine 2: End of Transfer

Reactivation of the Start I/O Routine

The actual data transfer has completed and the postprocessing activities are ready to begin. That is, the releasing of resources that were allocated in the first phase of the start I/O routine. This includes: releasing the map registers and the data path (Figure 26).

The interrupt routine transfers control to the start I/O routine in interrupt context and the start I/O routine immediately creates a fork process, thereby allowing other fork processes in the system to execute. This is done by a VAX/VMS routine: IOFORK. The IOFORK routine disables interrupts, saves registers R3 and R4, stores the driver program counter in a UCB field, transfers the driver to a fork queue, and finally, reactivates the interrupt service routine. The interrupt service routine then dismisses the interrupt by popping registers R0 through R5 off the interrupt stack and executing a return from interrupt instruction (Figure 25). The second phase of the start I/O routine is dequeued from the fork queue and executed as a fork process. The function of the second phase of the start I/O routine is illustrated in Figure 26. The EMR 760 never really requested a data path so there is no need to release one, but map registers were allocated and need to be released at this time. After releasing the map registers, the start I/O routine requests completion by a call to a VAX/VMS routine.

The operating system routine finishes the process by

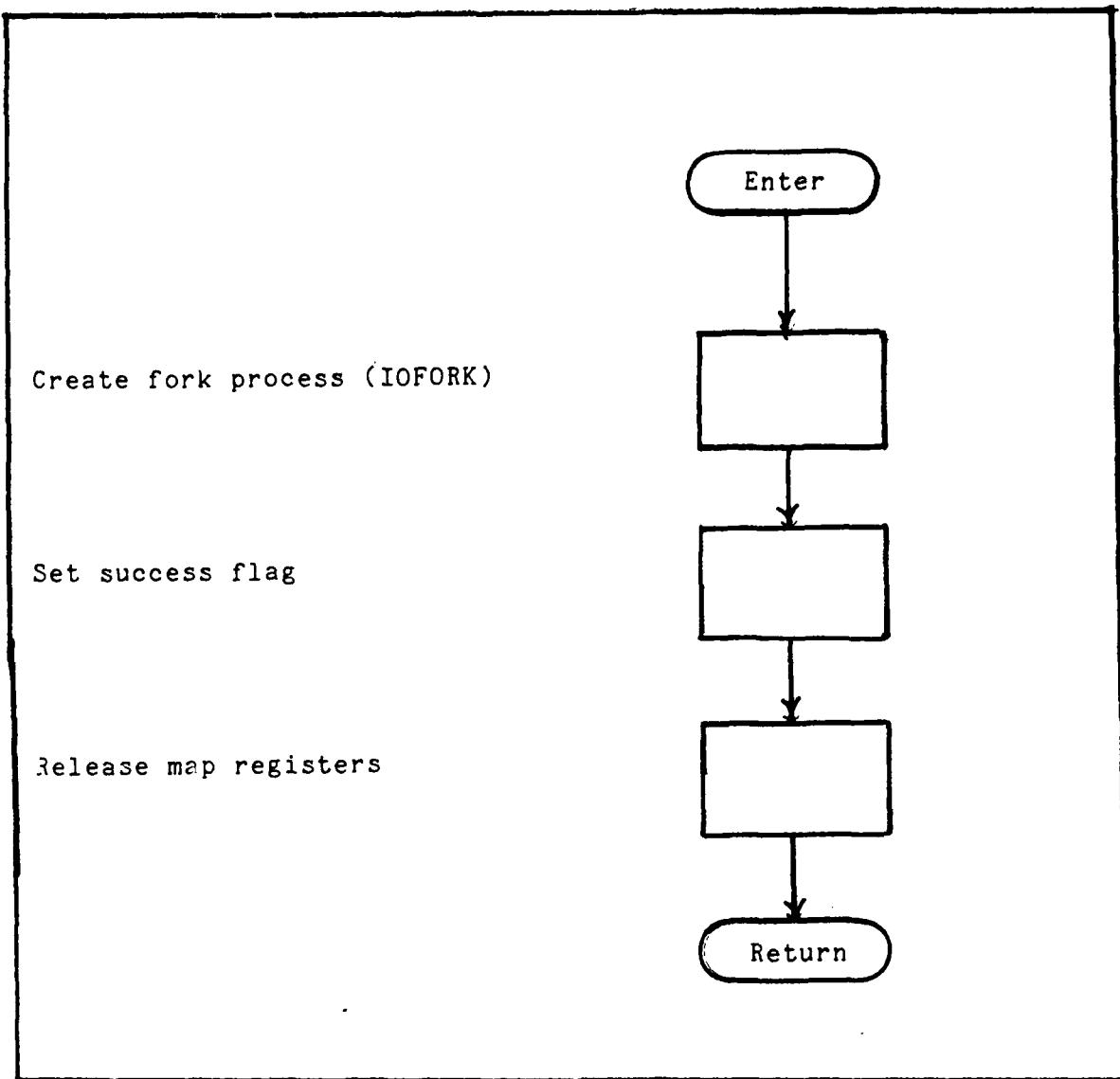


Figure 26. Start I/O Routine: Completion

inserting the I/O request packet into a postprocessing queue. Since it is more important to begin preprocessing, the operating system checks the preprocessing queue before allowing final postprocessing of the I/O request.

This completes the design of the EMR 760 driver, but there were several alternatives that justify discussion at this time.

Alternatives

There were several decisions made throughout the design of the driver. Two major alternatives are the requesting of a buffered data path and storing the device registers in UCB fields while in the interrupt routine.

The direct data path was used in the EMR 760 driver. The buffered data path provides a faster transfer rate. So, why use the direct data path? In the original design of the driver the buffered data path was used but had to be scrapped because of complications that are discussed in chapter V of this report. If the buffered data path was to be included in the driver, the following changes would be made. Before requesting map registers in the start I/O routine, a request would be made for a buffered data path, via the REQDPR macro. If this was included in the driver, the data path would need to be purged and released when no longer needed. This would be done immediately after the IO-

FORK macro in the start I/O routine. The other major alternative was to store device registers in UCB fields.

Storing device registers, while in the interrupt routine, is a technique used to pass information back to the user program. In the initial design the MA CTR and WC CTR were saved in UCB fields. Again, complications arose in the interrupt routine and all nonessential instructions were deleted from the interrupt routine. The MA CTR and WC CTR were to be used later in the start I/O routine to calculate the number of bytes transferred to the last buffer. This would be passed back to the user program via register R0.

These two alternatives were the only ones that resulted in a major design change of the driver. Other alternatives that were made quite early in the design include:

- * Selection of a software IPL. The software IPL could range from 8 through 11 (Ref 2:63). This controls the scheduling of the fork process and was chosen to be 8. An IPL of 8 is common for fork processes and this driver did not need extra consideration.
- * Selection of a hardware IPL. The hardware IPL could range from 20 through 23 (Ref 2:63). This controls the scheduling of the hardware interrupts and was chosen to be 23. This was necessary because the EMR 760 needs top priority when an inter-

rupt is requested. This will guaranty the immediate service of a hardware interrupt.

Summary

The design of the EMR 760 driver was produced in two phases. Initially the driver was designed using as many features of the VAX-11/780 that were necessary to achieve the optimal transfer rate across the UNIBUS. When the design failed upon use, it was replaced by a design that used the minimum features of the VAX-11/780 to achieve a DMA transfer. The next chapter discusses the application of both designs and the progress completed on each.

V. Verification and Validation

The purpose of this chapter is to define the methods used to test and debug the device driver. It discusses many of the problems encountered while debugging the driver and illustrates a typical application program that uses the driver.

Syntactic Errors

The debugging of the driver proceeded in two phases: the debugging accomplished at AFIT and the debugging performed on site at AFWL. The first phase consisted of syntactic error checking and driver initial compilation. The first phase was performed on the VAX-11/780 in the AFIT Engineering building. After several compilations, the driver was compiled and linked without any errors. The driver could not be loaded into the system at AFIT because there was not an EMR 760 attached to its UNIBUS. The next phase of debugging had to be performed on site at AFWL and consisted of a walkthrough and testing of the driver.

Walkthrough

The second phase of debugging was performed at the data

conversion branch of AFWL on their VAX-11/780. The second phase consisted of a program walkthrough with two computer professionals at AFWL, Ken Summers and Gene Simpson. The objective of the walkthrough was to inform the personnel at AFWL of the decisions taken in writing the driver, familiarize them with the routines and tables used in the driver, and to weed out design errors that were still in the driver.

Each section of code was discussed and possible errors or alternatives in that section were also discussed. The walkthrough lasted approximately two hours and proved very beneficial. During the walkthrough, several design errors were noted and resolved later. When all errors were resolved, the process of loading the driver into the system continued.

Loading the Driver

Loading the driver into the system appeared to be a very simple operation. The procedure for loading the driver into the system is explained in detail in chapter 15 of ref 5. These steps were followed and everything was proceeding accordingly, until the system needed the tr-value of the UNIBUS adapter that the device was attached to. The tr-value (transfer request value) is the SBI arbitration line to the UNIBUS adapter. SBI arbitration lines provide a priority control mechanism over the use of the SBI. After

many hours of fruitless efforts, another computer professional at AFWL; Lt Randy Rushe, who has a very comprehensive understanding of the VAX-11 system; was called upon to assist in this problem of finding a tr-value. Lt Rushe claimed the tr-value was 3 for a single UNIBUS system, which was the configuration of the VAX at the data conversion branch. This tr-value was later verified on page 128 of ref 2. With this tr-value, the process of loading the driver in the system proceeded without difficulties. Once loaded the driver needed to be tested. This was done with a very simple user program.

Testing the Driver

Testing of the driver proceeded with a simple user program. Figure 27 describes a flow diagram of this initial user program. The user program assigned an I/O channel for the EMR 760 and proceeded to call the driver using a system service routine (steps 2 and 3 of Figure 27). The program would then continuously display the first few locations in the buffer area of computer memory (steps 4 and 5 of Figure 27). This program was executed several times and on every call to the driver the system would generate a fatal error. At this point the driver needed to be debugged to find the cause of the fatal error.

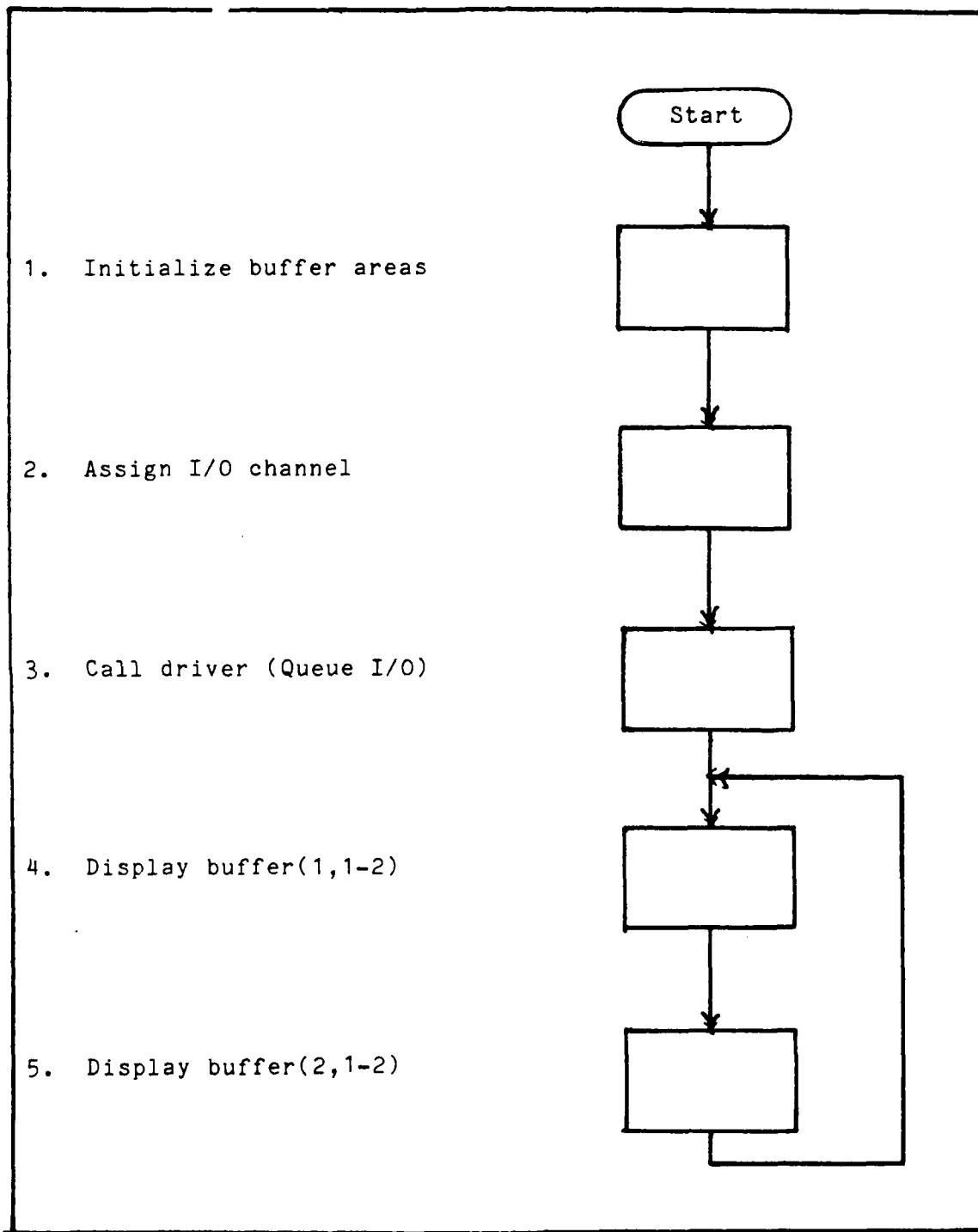


Figure 27. Initial user program

XDELTA

The technique to debug a device driver is explained in chapter 14 of "Guide to Writing a Device Driver." This technique for debugging the driver included the use of a DEC supplied debugger called XDELTA. DEC supplies two system debuggers as part of the software package with their VAX-11/780: DELTA and XDELTA. XDELTA is used to debug routines executing in the most privileged context of the system, that required by device drivers. Through the use of breakpoints and a stepping function, the cause of the error was found to be an instruction that used a device register as a 32-bit longword instead of a 16-bit word. One of the restrictions for code of a device driver is to access device registers using word oriented instructions (the UNIBUS is a 16-bit word oriented bus). The instruction that used a device register as a longword was the EXTZV #16,#2, R0, EM_MAX1(R4), see appendix A (Ref 3:199). This is not as obvious a longword instruction as a MOVL R1, R2; but upon closer examination of the addressing mode, the instructions required the source and destination to be longword addresses. After this and several similar instructions were corrected, the process of testing the driver continued. The corrected driver was loaded into the system and the user program, see Figure 27, was executed.

Selection of Data Paths

After the longword instructions were corrected, the user program began displaying the contents of a few buffer locations on the terminal. This didn't last long before the system generated a fatal error. Upon subsequent executions of the user program it was observed that these fatal errors were intermittent, they would not occur at the same time. Sometimes a few words were transferred and other times a whole block would be transferred. This was very puzzling and, once again, the services of Lt Rushe were called upon. Lt Rushe claimed the buffered data channels had always been a problem on his system when they were used to transfer more than 16 bits of data. Assuming this to be the problem, the driver code was revised to use the direct data path. This cured the intermittent fatal error. The device could transfer data to memory buffers without generating any system errors.

Interrupt Handling

The device, after initialization from the driver, began transferring data to the memory buffers successfully. This apparent success was short-lived for when an interrupt was stimulated the system would generate a fatal error. Using the same techniques described above, breakpoints were placed in the interrupt routine. If the system ever entered the

AD-A100 777

AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH 54500--ETC F/G 9/2
WRITING A SOFTWARE PACKAGE FOR AN EMR 760 BDC ON THE VAX-11/780--ETC(1)
DEC 80 D L RALL
UNCLASSIFIED AFIT/GCS/EE/80D-13

ML

2 OF 2
200717

END
DATE
FILED
7-81
DTIC

routine, XDELT A would take over and the routine could be stepped through to find the cause of the error.

When an interrupt was stimulated, XDELT A received control in the device interrupt routine. Using the stepping function, the interrupt routine was executed one instruction at a time. Everything appeared in order and the interrupt routine jumped to the suspended driver start I/O subroutine, which was waiting for an interrupt or timeout. The first instruction in the reactivated start I/O routine (IOFORK) is a macro call to a VAX/VMS routine which creates a fork process from the remainder of the start I/O routine. The function of the IOFORK routine is to create a fork process of the remainder of the start I/O routine and transfer control back to the interrupt routine at the instruction following the JSB, which had called the start I/O subroutine. Using XDELT A, the IOFORK routine was stepped through and transferred control back to the interrupt service routine. The interrupt routine continued executing until the REI (return from interrupt or exception, Ref 3:311) instruction was encountered. Upon execution of the REI instruction the system generated a fatal error. It seemed very peculiar for the REI instruction to generate an error; but the REI instruction does an extensive amount of error checking and there could be a number of reasons for it to fail (Ref 2:58). The most obvious error would be in the integrity of the Interrupt Stack (IS). If the IS had been tampered with, the return address necessary for the REI to transfer control might

be destroyed or misplaced. With this in mind, the interrupt routine was executed once again and was stopped just before the REI instruction. The IS was interrogated and the return address on the stack was to a VAX/VMS routine called NULL\$PROCESS. The return address should have been to the process executing before the interrupt was generated: the user program (Ref 2:58). Several attempts were made to resolve this problem. First all unnecessary instructions were removed from the interrupt routine. This included the instructions to store the device registers into UCB fields. With all unnecessary instructions removed the driver was tested again. The system still generated a fatal error on the REI instruction. The next attempt to alleviate this problem was successful.

Since the problem appeared to be in the switching from an interrupt context to that of either a fork process or user process, the call to reactivate the start I/O routine was removed. With this deleted, the interrupt service routine could dismiss the interrupt but the driver would always be waiting to be reactivated (or a timeout). The disadvantage of not reactivating the driver was the user program could not complete until the driver was finished with I/O processing. The system had to be rebooted to start the process over again. Removing the JSB instruction cured the fatal error but left the driver in a "jerry-rigged" environment. A considerable amount of time had been expended and no solution had been found. So, with time running short,

the testing of the driver proceeded. The next course of action was to verify the data transferred to memory.

Simulation

A set of sample data was used to verify the integrity of data transferred to memory. The sample data consisted of twenty 16-bit words. The set of sample data was then fed as continuous input to the EMR 760, see Figure 28. An external start signal was sent at the beginning of each set to assure the first word across the UNIBUS was the first data word of the sample set. After the sample data was transferred to memory, the buffers were interrogated. It was observed that the memory buffers contained the corresponding values of the sample set. The integrity of the data was maintained. With the integrity of the data maintained, the next step was to vary transfer rates in an attempt to find the maximum transfer rate.

Application Program

A new users application program had to be designed to accomadate error checking and allow transfer of the buffers to a file on disk. This program would be designed to be the final user program and would be used in subsequent applications of the EMR 760. A flow diagram of this program is il-

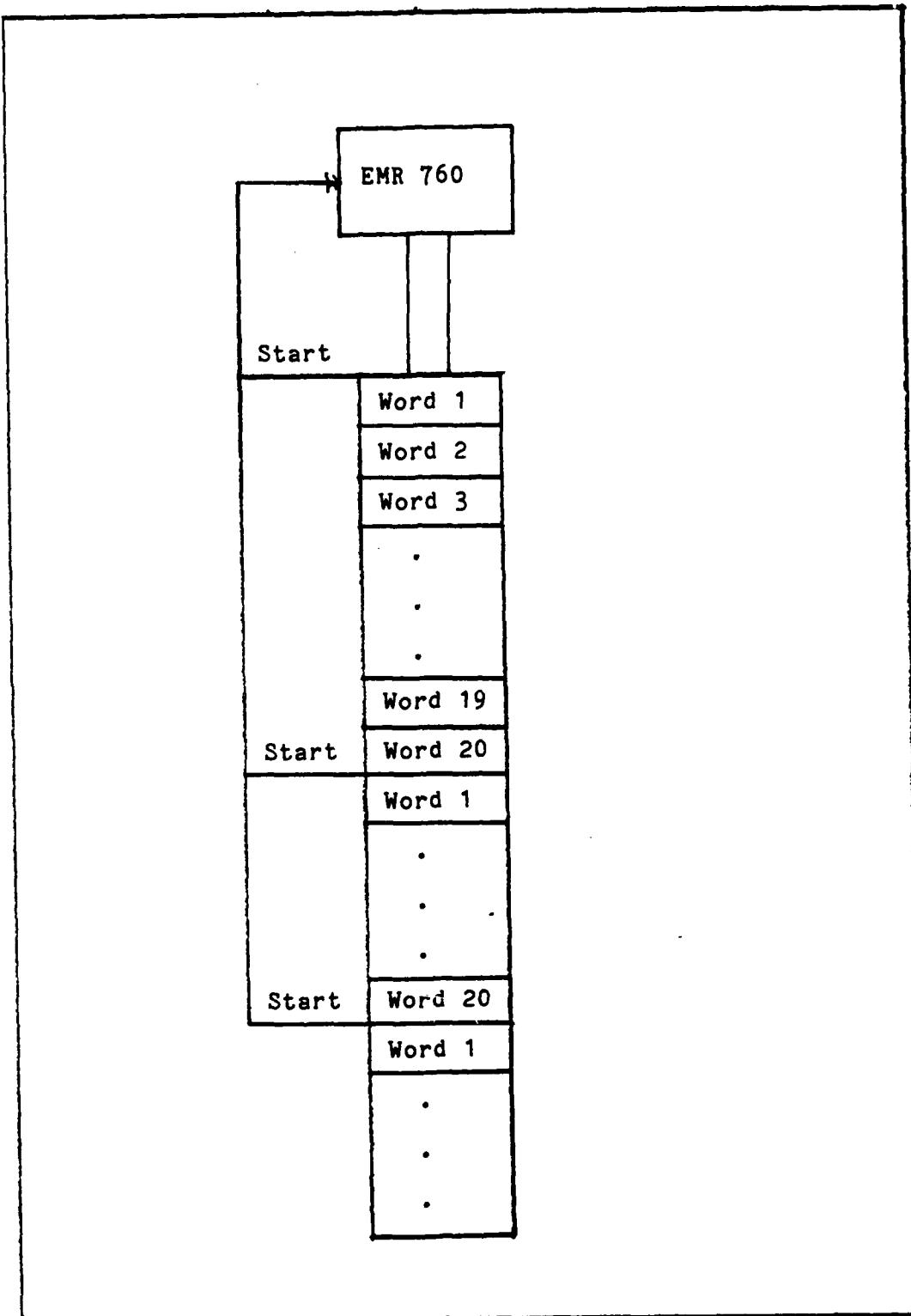


Figure 28. Continuous Sample Data

lustrated in Figure 29 with a listing in appendix B.

This program would request all the necessary channels, initialize buffer areas and constants, create a file on disk, and call the driver (steps 1 through 5 of Figure 29). At this point the program would poll device control registers and when the EMR 760 was finished filling buffer 1, buffer 1 would be transferred to a file on disk (steps 6 through 9). The disk parameters would then be adjusted so buffer 2 could be transferred as soon as the EMR 760 was finished with it (step 10). The program would then poll device registers again until the EMR 760 was finished with buffer 2. Buffer 2 would be transferred to disk as long as the disk was finished transferring buffer 1. If the disk wasn't finished transferring buffer 1 then the EMR 760 is operating at a rate faster then the disk and there would be the possibility of overlapping the data. If this condition ever occurred (the EMR 760 operating faster than the disk) a message would be displayed on the operators console and the program would terminate. If everything operated in order, the disk parameters would be adjusted to transfer buffer 1 to disk the next time around (step 16 of Figure 29). This series of operations would continue until, while polling the control registers, control register one informed the program that the EMR 760 had been turned off (step 17 of Figure 29). If the EMR 760 had been turned off, the program would evaluate the EMR 760's word count counter for the total number of bytes transferred to the last buffer.

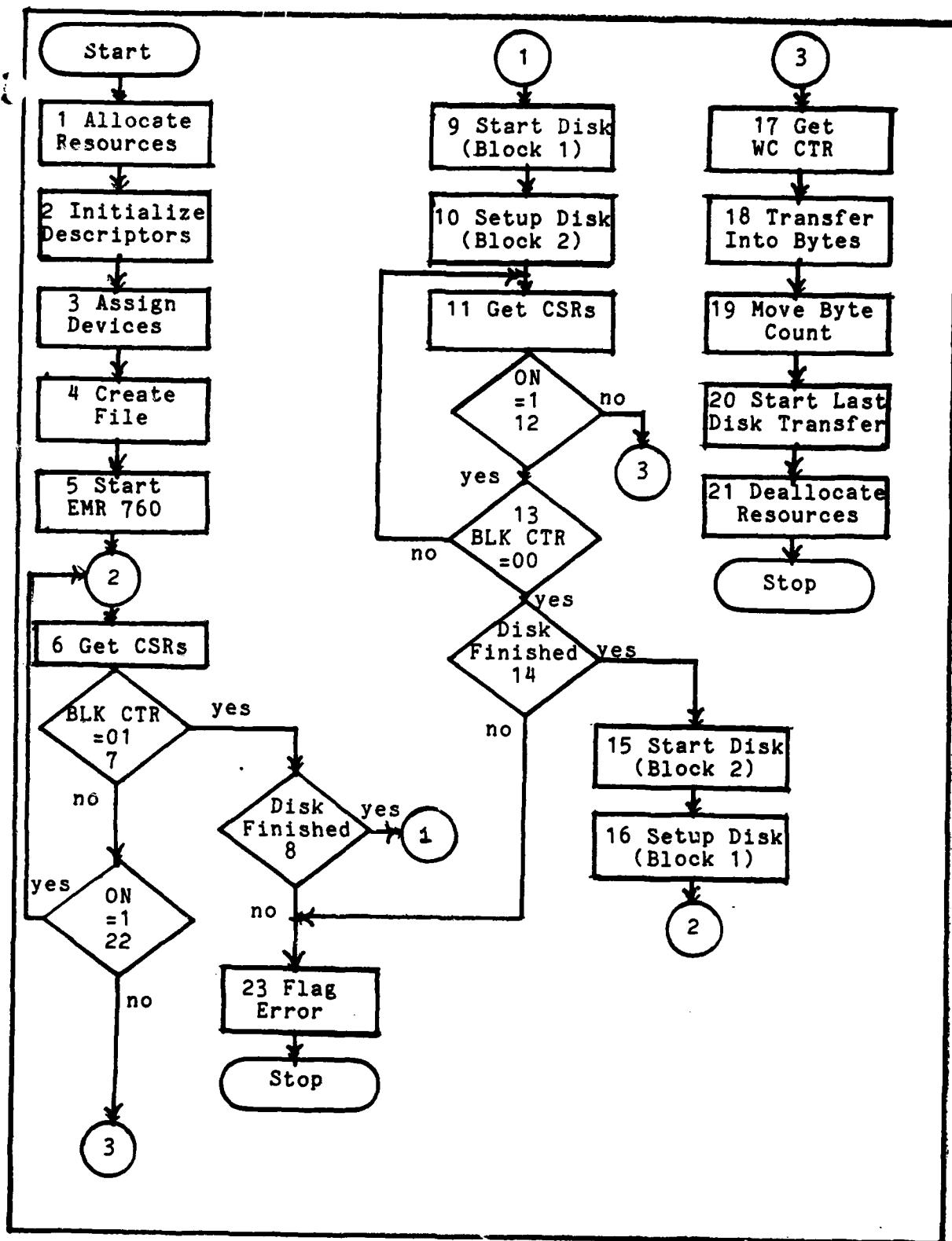


Figure 29. Application Program

The program would use this byte count as a parameter in the final call to write to the disk. After the transfer of those bytes to disk, the program would display the total byte count transferred. The byte count was being updated every time a memory buffer was being transferred to disk. Since the program checks to make sure the EMR 760 isn't going too fast, it is very easy to find the maximum transfer rate.

Transfer Rates

Using the program described above, the transfer rate can be increased until the program displays an error message indicating the EMR 760 is operating to fast. Several data runs were performed and the results are described in Table IV. Table IV shows several transfer rates, the number of seconds the EMR 760 was turned on, and the total byte count the program displayed. This provides two pieces of information. First, it gives some idea as to how long the total transfer session will be. Second, it verifies the correctness of the program. Using the transfer rate and the number of seconds the EMR 760 was turned on, the total byte count could be computed (see column four of Table IV). Comparing this with the programs total byte count could give some justification as to the validity of the whole operation. The figures are compared in column 6 of Table IV and appear to

be well within the margin of error associated with the timer (watch) used. Unfortunately, the program never encountered a condition where the EMR 760 ran too fast for the disk. This is because the simulator, which inputted data to the EMR 760, could input no faster than 190,000 w/s. The maximum transfer rate with the disk used, RP06, was over 350,000 w/s (Ref 6:5-1). So, it seems very plausible that the EMR 760 never transferred at a rate faster than the disk. An interesting note arises from Table IV. Namely, on the run where the EMR was transferring at 190,000 w/s and was allowed to transfer for 60 seconds, the total number of bytes transferred was approximately 23,000,000 bytes. This is interesting because the file on disk had only 10,000,000 bytes allocated. The system should have given an error but none occurred. No further study was done to examine the consequences of this action. With very little time (and TDY money) left, the problem of the interrupt routine was given further consideration.

Success

The last actions taken on the interrupt routine were to delete unnecessary instructions and remove the JSB instruction that reactivated the start I/O routine. So, the first thing was to reinsert the JSB instruction to call the start I/O routine. With this added, the driver was reloaded into the system. The new application program was executed. It

started the transfer operation and an interrupt was stimulated. The driver proceeded to finish I/O processing without complications.

This concludes the discussion of testing and debugging. There were several aspects of the driver that deserved further testing, but there was too little time allocated to the testing stage of the device driver.

Summary

The testing of the driver was done at AFIT and AFWL. Throughout the testing of the driver, several things were eliminated from the design because of complications. The buffered data path was eliminated because of unreliability. The instructions to transfer device registers to UCB fields were removed to simplify the interrupt routine. With these modifications, the driver executed successfully. The next chapter discusses conclusions and recommendations of this research.

Table IV. Verification of Transfer Rate

Transfer Rate (KW/S)	Seconds (S)	Bytes Bytes (B/W)	Expected (KB)	Actual (KB)	Percentage Difference (%)
10	45	2	900	896	0.44
100	20	2	4000	4154	3.85
190	10	2	3800	4058	6.79
190	60	2	22800	23258	2.00

VI. Conclusions and Recommendations

This chapter addresses conclusions and recommendations based upon the status of the current research. Topics to be discussed include: integration of an EMR 760 into a VAX-11 configuration; optimization of transfer rates; and degradation of system performance.

Integration

The integration of the EMR 760 into a VAX-11 environment needs to be discussed because of the conflict between the VAX-11 and the EMR 760 operations. The VAX-11/780 performs very well in a time-sharing environment. The VAX-11/780 will also perform very well in a dedicated time-critical environment. The VAX-11/780 will perform fairly well when given both environments (time-sharing and time-critical). Using the VAX-11/780 for both applications will obviously degrade the system performance for both environments. This is because of the overhead required by the operating system to synchronize the two environments. A good system configuration would involve the use of a front end microprocessor (Figure 30) to perform the data acquisition task. The front end microprocessor would relay the data to the VAX-11/780 via a shared disk. The microprocessor, PDP-11, would receive data in the same manner and on

the same bus, UNIBUS, that the VAX-11/780 was receiving it. This means that the PDP-11 would complete the task sooner than the VAX-11/780 because it does not have the operating system overhead needed to control two environments and it does not need access to shared resources, UNIBUS lines and SBI lines. With this configuration (Figure 30), the VAX-11/780 could provide a very fast and efficient time-sharing system. The VAX-11/780 could also perform any processing that was needed by the data. Figure 30 also illustrates the ease of expansion using this approach to the data acquisition problem. As can be seen, a multiprocessor system could be very easily adapted to the overall system configuration and all the processors could be connected in a network environment (Figure 31).

The cost of a PDP-11 could be offset from the cost of the memory needed to run the configuration on the VAX-11 system. The expandability is much cheaper with the PDP-11. When several EMR 760s are expanded to the system, the need for memory increases tremendously. Each EMR 760 running will have to have two memory buffers allocated. Each EMR 760 running will have to have 128 map registers allocated if the memory buffers are 16K words long. Thereby allowing only three active EMR 760s per UNIBUS or a maximum of 12 active EMR 760s per VAX-11/780. Another computer would need to be purchased to allow thirteen active EMR 760s, a \$300,000 investment. With the system illustrated in Figures 30 or 31, the investment would only be \$20,000.

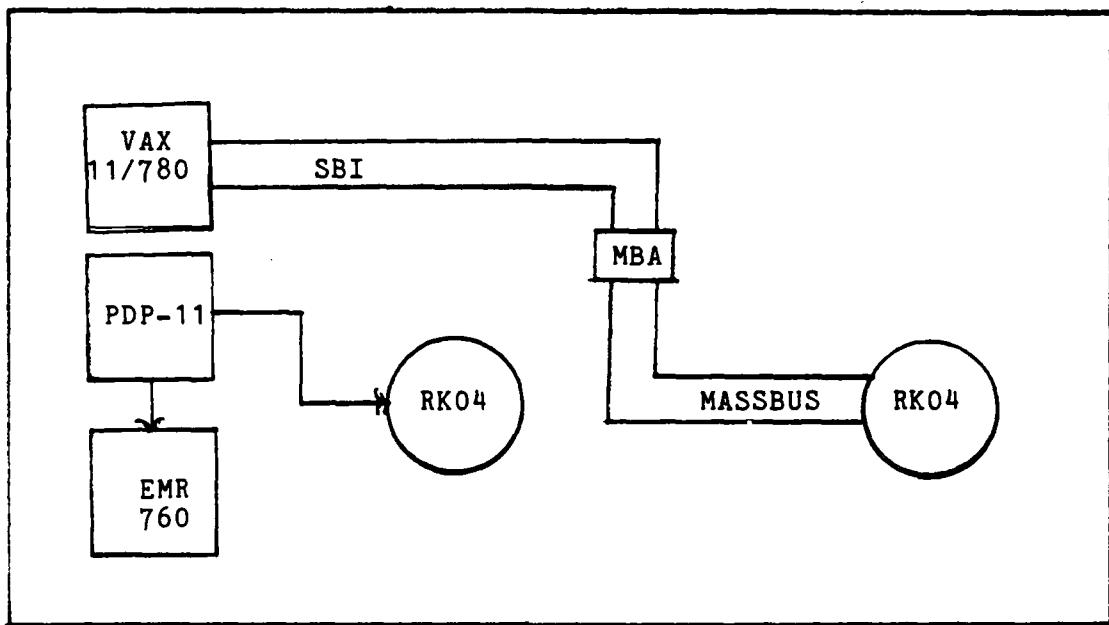


Figure 30. Front End Microprocessor

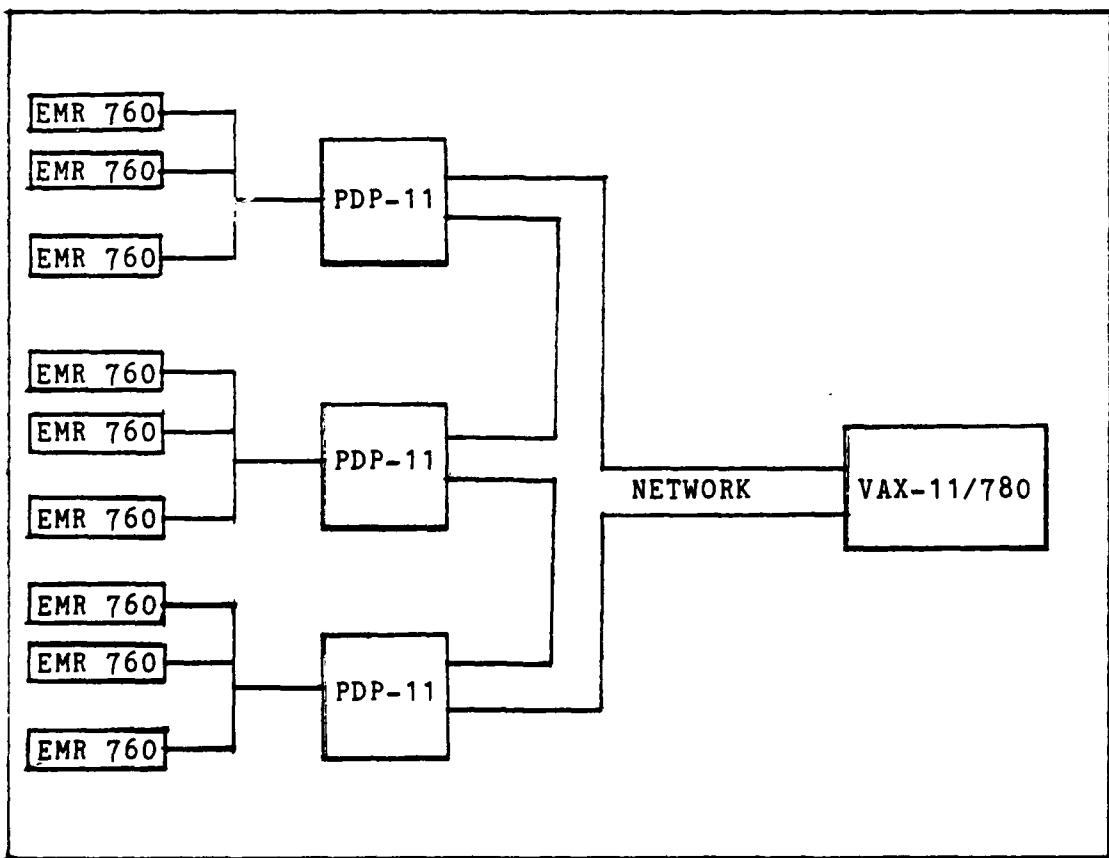


Figure 31. Network Configuration

This is a very simplistic example and does not consider all costs associated with both systems: extra disks, UNIBUS adapters, and other related peripherals. A recommendation for further research is definately in order. Evaluation of the current system and proposed systems should be researched with a recommendation to the optimal configuration. Another consideration for an optimal system is the throughput. The following section discusses the transfer rate obtained on the VAX-11/780.

Transfer Rates

The transfer rate between the EMR 760 and the VAX-11/780 was a minor consideration of the research. The maximum transfer rate for the driver was never achieved on the simulated system but was found to be greater than 190,000 words per second. This was achieved on the slowest channels of the VAX-11/780: the direct data path and the UNIBUS.

The direct data path is the slowest path on the UNIBUS. It can maintain a maximum transfer rate of 750,000 bytes per second (375,000 words per second (Ref 2:179)). To achieve an optimal transfer rate the buffered data paths would be used. These data paths allow a transfer rate from 1,170,000 bytes per second through 1,350,000 bytes per second (Ref 2:186). It was noted earlier that buffered data paths were

not used because of an apparent problem in transferring data over them. DEC still maintains that there is nothing wrong with the data paths and, when used correctly, will operate without error. Another recommendation for further study would be in the buffered data paths and if they really are as reliable as DEC claims.

Another "bottleneck" in the transferring of data is the UNIBUS. The UNIBUS is the slowest bus on the VAX-11/780 and allows a maximum throughput rate of 1,500,000 bytes per second for one through four bytes (Ref 2:12). The other bus structures on the VAX-11/780, MASSBUS and SBI, allow respective transfer rates of 2,000,000 bytes per second and 13,300,000 bytes per second. The ideal connection would be directly to the SBI bus. This would only be feasible for a multiunit environment with the EMR 760, because, the EMR 760 can transfer at a sustained rate of 1,000,000 bytes per second (Ref 5:5-1). So, with thirteen devices attached to the SBI, the maximum throughput rate for the SBI would be optimized.

Finally, the EMR 760 provides several modes of operation and only the simplest were chosen for this research. Further research into the optimal transfer rate of the EMR 760 would be beneficial. This would include using adaptive burst mode with a FIFO length of 14. Several modes could be given intensive testing with the final outcome of the best mode for two types of environment: a single EMR 760 and

multiple EMR 760s. The EMR 760 can use a maximum of four buffer areas, only two were used here. An analysis of this approach could be performed and a recommendation for the number of buffers and their size could be made.

Degradation of System

The VAX-11/780 was a poor choice for this data acquisition task. The time-sharing environment will be drastically affected by the integration of the EMR 760s. An analysis of the actual degradation would be very beneficial. The analysis could prove or disprove the need for a front end processor.

The application program that calls the EMR 760 driver is the main reason for the degradation of the time-sharing environment on the VAX-11/780. Some recommendations on research for a more efficient method of control is provided here.

Application Program

The application program used to call the driver needed total dedication from the VAX-11/780. The reason being the detection of the EMR 760 changing data transfers from buffer 1 to buffer 2 or visa versa. This detection was necessary for the program to know when to send the buffer areas to a

file on disk. There are several methods that will allow the control needed by the program.

One method would allow the use of interrupts and after a block end allow the EMR 760 to request an interrupt. In the interrupt service routine some sort of flag would be set to give the program the necessary information to control the sending of the buffer areas. The problem with this method is the amount of time taken by the operating system to transfer control to the interrupt routine. The time needed by the operating system to transfer control to an interrupt routine was found to be anywhere from 3 to 10 microseconds (Ref 5). This would put a significant constraint on a single EMR 760 configuration, but if many EMR 760s were configured in the system, it would not seem to be a significant disadvantage.

Another method of control that would require limited modification of the existing program would use a call to a routine that would suspend the program for two-thirds of the time needed to fill up one buffer. For the buffer size used here and a maximum throughput rate of 300,000 words per second, it would take the EMR 760 0.05461 seconds to fill one memory buffer. The program could suspend itself for 0.036 seconds. This would allow other users access to the system while the EMR 760 was filling the memory buffers. A problem that would need research is the reliability of wake up. In other words, will the system reactivate the program before

the EMR 760 finishes filling the memory buffer?

A final method of control uses the input data as the method for synchronization of the buffer areas. If the data was assured to be nonzero, the beginning location of the buffer areas could be interrogated until they became nonzero. The buffer would be output to disk and the beginning location could be cleared. This is no better than polling the device registers because the program would still need complete dedication by the computer. The advantage is the device register addresses do not need to be known. This is a real problem because the address assignment to the device registers may vary upon subsequent connection of the device driver. If the address assignment does change, the application program used now will not work. When the driver was loaded into the system the driver data base was interrogated for the physical location of control register one. A very worthwhile study would find the method used by VAX/VMS in placing the driver data base into the system I/O data base. The application program could then interrogate the UCB of the device for the physical location of control register one.

Summary

The amount of research performed would vary upon the time given to each topic and the number of topics that could be covered. All topics would be worthwhile not only to the student, who would learn a great deal because of the sophis-

tication of the VAX-11/780, but also to the data conversion branch at AFWL.

Bibliography

1. Digital Equipment Corporation. VAX11 Software Handbook. Description of VAX-11/780 Software. Digital's Sales Support Literature Group, 1977.
2. Digital Equipment Corporation. VAX11/780 Hardware Handbook. Introduction to VAX11 Hardware Elements. Digital's Sales Support Literature Group, 1979.
3. Digital Equipment Corporation. VAX11 Architecture Handbook. Introduction of the Addressing Modes and Instruction Set of the VAX11. Digital's Sales Support Literature Group, 1979.
4. Digital Equipment Corporation. VAX/VMS Guide to Writing a Device Driver. System Manual. Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754, March 1980.
5. EMR Telemetry. EMR Model 760 Buffered Data Channel. Product Specification. EMR Telemetry, PO Box 3041, Sarasota, Florida 33578, March 1979.
6. Digital Equipment Corporation. VAX11/780 Technical Summary. System Document. Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754, 1978.
7. Jensen, Randolph and Charles Tonies. Software Engineering. Englewood Cliffs: Prentice-Hall, Inc., 1979.

Appendix A: EMR 760 Driver

```
.TITLE EMDRIVER - EMR 760 DEVICE DRIVER
.IDENT /V02/1/
;
; The EMR 760 is a high-speed, DMA buffered data channel.
; It can be used to input to or output from main memory. This
; driver uses only the input option on the EMR 760. The user
; program calling this driver should have defined two contiguous; p
; buffers of memory with a total length of 32K (32 * 1024) words.
; The EMR 760 will continuously fill these buffers until it has
; finished transferring data.
; This driver uses the UNIBUS direct data path to transfer data to
; memory. It allocates 128 UNIBUS map registers to access the 32K
; words of memory. A maximum transfer rate of 750,000 sixteen bit
; words is possible with a realistic aggregate rate of 500,000 words
; for a long transfer period (several megabytes of data).
.page
.SBTTL EXTERNAL AND LOCAL SYMBOL DEFINITIONS
;
; External Symbols:
; Most external symbols have the form TYP$D_VAR
; Where:
; TYP is the letters following the $ and preceding
; the DEF letters in the macro expansions below
; D is one of (V,B,W,L,Q). This defines the variable
; length. (V-Bit,B-Byte,W-Word,L-Long,Q-Quad)
; VAR is the variable name of the offset needed.
;
;
$SSDEF      ;condition codes
$CRBDEF    ;CHANNEL REQUEST BLOCK OFFSETS
$DDBDEF    ;DEVICE DATA BLOCK OFFSETS
$IDBDEF    ;INTERRUPT DATA BLOCK OFFSETS
$IODEF      ;I/O FUNCTION CODES
$IPLDEF     ;HARDWARE IPL DEFINITIONS
$IRPDEF     ;I/O REQUEST PACKET OFFSETS
$UCBDEF     ;UNIT CONTROL BLOCK OFFSETS
$VECDEF     ;INTERRUPT VECTOR BLOCK OFFSETS
$DCDEF      ;DEVICE CLASS DEFINITIONS
$DEVDEF     ;DEVICE TYPE DEFINITIONS
$UBADEF     ;UNIBUS ADAPTOR BLOCK OFFSETS
;
; Argument List (AP) Offsets:
; These are the input parameters from the call to the
; $QIO system service in the user program. P1 is the
; starting address of the memory buffer. P2 is the byte
; count of the memory buffer. This will always be 64K.
; Actually there are two memory buffers, but since they
; must be contiguous it can be considered one buffer.
; The other parameters aren't used and are placed here
; for clarity and standardization.
```

```

;
P1 =0
P2 =4
P3 =8
P4 =12
P5 =16
P6 =20
;
;
;
;
; Other Constants
;
DEFAULT_BUFSIZE=16383      ;word length minus one of a memory
                            ;buffer. This will be used to initialize
                            ;the word count registers in the EMR 760.
EM_START_CSR=^X8F      ;This is the value to be written into control
                        ;register 1. It defines the following operation
                        ;Input, Cyclic mode with two buffers and use
                        ;the external start as the means to start the
                        ;EMR 760.
EM_TIMEOUT=65535      ;time out in seconds. The timeout should
                        ;never occur and is of no concern to this
                        ;driver, but VAX/VMS routines need it.

.PAGE
.SBTTL EMR-760 DEFINITIONS
;
.PAGE
.SBTTL DEVICE REGISTER OFFSETS
;
; Device register offsets that use control register one as a base
;
$DEFINI EM
;
; Current location counter is always zero after the $DEFINI macro
; Since the Control/status register 1 is the 14th device register
; and it should be considered the base, the location counter is
; moved back 13 words or 26 bytes. This can be done by either
; .=-26 or .=-26 because .=0 at this point. EM_MA1 through EM_MA4
; are the memory address registers. They will be used to hold 16
; of the 18 bits needed for a UNIBUS address for their respective
; buffer areas. EM_MAX1 through EM_MAX4 will contain the 2 bit
; extension to the UNIBUS address of buffers 1 through 4. This
; driver will only use EM_MA1, EM_MA2, EM_MAX1, and EM_MAX2
; because only two buffers are used. EM_WC1 through EM_WC4
; are the length of their buffer areas minus one. Again, this
; driver only uses EM_WC1 and EM_WC2. This macro expansion only
; defines the byte offset that a particular device register is from
; control register one(EM_STATUS1). EM_MA1 will be equivalent to
; -26 throughout the driver. Similarly for the rest.
.=.-26

$DEF      EM_MA1  .BLKW 1
$DEF      EM_WC1  .BLKW 1
$DEF      EM_MA2  .BLKW 1
$DEF      EM_WC2  .BLKW 1 ;

```

```

$DEF      EM_MA3 .BLKW 1 ;
$DEF      EM_WC3 .BLKW 1 ;
$DEF      EM_MA4 .BLKW 1 ;
$DEF      EM_WC_CTR
          .BLKW 1 ;
$DEF      EM_MAX1 .BLKW 1

.=-2
$DEF      EM_MA_CTR      ;
          .BLKW 1
$DEF      EM_MAX2 .BLKW 1
$DEF      EM_MAX3 .BLKW 1 ;
$DEF      EM_MAX4 .BLKW 1 ;
$DEF      EM_DOR .BLKW 1
$DEF      EM_STATUS1
          .BLKW 1 ;
$DEF      EM_STATUS2
          .BLKW 1 ;

$DEFEND EM
;
.PAGE
.SBTL STANDARD DRIVER TABLES
;
; The Driver Prologue Table (DPT) is used to define characteristics
; of the driver to the operating system. The operating system uses
; the end label to calculate the size of the driver. It also likes
; to know the type of adapter, size of the UCB(default for this
; driver), and name of the driver.
;
DPTAB  -
END=EM_END,-
ADAPTER=UBA,-
UCBSIZE=<UCB$K_LENGTH>,-
NAME=EMDRIVER
;
; The Driver Prologue Table Store(DPT_STORE) macro is used to
; initialize fields in the driver data base. The DPT_STORE
; commands that follow the DPT_STORE INIT and precede the
; DPT_STORE REINIT are used to initialize fields when the driver
; is initially loaded into the system. The commands from the
; DPT_STORE REINIT to the DPT_STORE END are used to initialize
; fields when the driver is reloaded into the system(usually
; these consist of routine addresses that could have been changed
; by a newer version of the driver).
;
DPT_STORE INIT
;
DPT_STORE UCB,UCB$B_FIPL,B,8
DPT_STORE UCB,UCB$L_DEVCHAR,-
          L,<DEV$M_IDV>
DPT_STORE UCB,UCB$B_DEVCLASS,-
          B,DC$_SCOM
DPT_STORE UCB,UCB$B_DIPL,B,22
;

```

```

;
; DPT_STORE REINIT
;
; DPT_STORE DDB,DDB$L_DDT,D,EM$DDT
; DPT_STORE CRB,CRB$L_INTD+4,-
;     D,EM_INTERRUPT
; DPT_STORE CRB,CRB$L_INTD2+4,-
;     D,EM_INTERRUPT2
; DPT_STORE CRB,CRB$L_INTD+4,-
;     D,EM_INTERRUPT
; DPT_STORE CRB,CRB$L_INTD+VEC$L_INITIAL,-
;     D,EM_CONTROL_INIT
;
; DPT_STORE CRB,CRB$L_INTD2+VEC$L_INITIAL,-
;     D,EM_CONTROL_INIT
;
;
; DPT_STORE END
;
;
; Driver Dispatch Table (DDT): The DDTAB macro is used to define
; the device name, start I/O address and the Function Decision
; Table (FDT) address.
;
; DDTAB DEVNAM=EM,-
;     START=EM_START_IO,-
;     FUNCTB=EM_FUNCTABLE
;
;
; Function Decision Table: This table defines the functions allowable
; for this device and the routines to enter for the particular
; functions. The first macro expansion of FUNCTAB defines all
; functions allowable. The second expansion of FUNCTAB defines
; the buffered I/O functions allowable(this driver uses no
; buffered I/O functions). The third and following expansions of
; FUNCTAB lists functions and routines to be executed for that
; function. The EMR 760 is being used to input to a users buffer area
; and is therefore considered a read operation. The +EXE$READ is
; a VAX/VMS routine that performs device-independant initialization
; for a read function.
;
; EM_FUNCTABLE:
;     FUNCTAB ,<READVELK,READLBLK,READPELK>
;     FUNCTAB ,
;     FUNCTAB +EXE$READ,-
;             <READVELK,READLBLK,READPELK>
;
; .PAGE
; .SBTLL CONTROLLER INITIALIZATION ROUTINE
; EM_CONTROL_INIT:
;     This routine readies the controller for I/O operations.
;     It is executed when the driver is loaded into the system. Since
;     this is a dedicated controller, only one device attached, all

```

```

; unit initialization will be done in this routine also.

; Functional Description:
; Assigns the owner of the controller to the only device UCB
; allocated to this controller. Sets the status of the device in
; the UCB block to online. Clears both device registers.

; Inputs:
; R4 address of EM_STATUS1(Control register 1)
; r5 address of IDB
;

EM_CONTROL_INIT:
    MOVL    IDB$L_UCBLST(R5),R0
    MOVL    R0, IDB$L_OWNER(R5)      ;only one owner
    BISW    #UCB$M_ONLINE,-        ;set device online
            UCB$W_STS(R0)
    CLRW    EM_STATUS1(R4)         ;clear control registers
    CLRW    EM_STATUS2(R4)

;

; RSB

;

.PAGE
.SBTTL START I/O OPERATION

; EM_START_IO:
; The start I/O routine is the workhorse of the driver.
; It is responsible for requesting, initializing, and releasing
; all resources the driver will need to transfer data from the
; EMR 760 to two memory buffers of 16K words each.
;

; Functional Description:
; This routine starts a DMA read operation. It loads UNIBUS
; map registers, calculates the physical starting address of
; both memory buffers used in the transfer, checks for a power
; failure and activates a transfer session by writing to the
; control registers in the EMR 760.
;

; Inputs:
; R3 address of I/O request packet
; R5 address of device's UCB
;

EM_START_IO:
    MOVL    UCB$L_CRB(R5),R4      ;get CRB address
    MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4
                                ;get Control register 1 address
    CLRW    UCB$W_DEVSTS(R5)      ;clear device status in UCB
    REQMPR                         ;request map registers
;

; Map registers are allocated using the byte count and the
; byte offset words in the UCB block.
;

;

; Load UNIBUS map registers
;

    LOADUBA

```

```

;
    MOVZWL UCB$W_BOFF(R5),R0      ;calculate starting address
                                    ;of buffer 1
    MOVL   UCB$L_CRB(R5),R1      ;get CRB
    MOVZWL CRB$L_INTD+VEC$W_MAPREG(R1),R2
                                    ;Get number of first map register allocated. There
                                    ;are 496 map registers. Only 128 are used by this driver
    INSV   R2,#9,#9,R0          ;Put map register number in bits
                                    ;9-18 of R0(bits 0-8 contain byte
                                    ;offset)
    EXTZV  #16,#2,R0,R3          ;Extract extension bits 17 _ 18
    MOVW   R3,EM_MAX1(R4)        ;Put in memory extension register
    MOVW   R0,EM_MA1(R4)        ;Put in memory address register
    ADDL2  #64,R2                ;Add 64 to the map register number
                                    ;(map registers are allocated contiguously so the
                                    ;64th map register that was allocated for this driver
                                    ;will point to the beginning of memory buffer 2).
    INSV   R2,#9,#9,R0          ;Proceed in same fashion for buffer 2.
    EXTZV  #16,#2,R0,R3
    MOVW   R3,EM_MAX2(R4)
    MOVW   R0,EM_MA2(R4)
    MOVW   #DEFAULT_BUFSIZE,EM_WC1(R4) ;both word counts should
    MOVW   #DEFAULT_BUFSIZE,EM_WC2(R4) ;be 16383 (16K - 1)
;
; Disable interrupts and check for power fail
;
    DSBINT
    BBSC   #UCB$V_POWER,-
            UCB$W_STS(R5),-
            POWER_FAIL
;
; If no power fail then activate the EMR 760
;
    MOVW   #EM_START_CSR,-
            EM_STATUS1(R4)
;
; Wait for a timeout or interrupt. Timeout should never occur.
; In any case the Driver will suspend itself and release control
; of the CPU. This allows the user program to continue from
; the point of the call to the $QIO system service routine.
;
    WFIKPCH TIMEOUT,#EM_TIMEOUT
;
; Now that an interrupt has been serviced go ahead and make this a
; fork process. No reason to do any more computation so request
; completion.
;
    IOFORK
    BRW    COMPLETE_IO
;
; If there was a power fail then set user status and depart

```

```

;
;POWER_FAIL:
    ENBINT
    MOVZWL #SS$_POWERFAIL,R0
    BRW    COMPLETE_IO
;
;
; Timeout should never be entered but if it does: clear control
; registers, create fork process, and give user as much info
; as possible.
;
;TIMEOUT:
    IOFORK
    CLRW   EM_STATUS1(R4)
    SETIPL UCB$B_FIPL(R5)
    MOVW   #SS$_TIMEOUT,R0
;
; REQCOM takes care of all I/O postprocessing
;
;COMPLETE_IO:
    REQCOM
;
    .PAGE
    .SBTTL INTERRUPT SERVICE ROUTINE
;
;EM_INTERRUPT:
;    This interrupt service routine is used to service a request
; on interrupt 1. The EMR 760 has the capability to request two
; distinct interrupts. Interrupt 1 is dedicated to signal a block
; end. Interrupt 2 is used to signal a user event that needs ser-
; vicing(end of transfer period). During normal operation this
; routine should not be entered. If it is just clear control
; register one and dismiss the interrupt. This will be plenty
; of notification to the user program since it polls control
; registers.
;
;Input:
;    0(SP)    pointer to address of IDB
;    4(SP)-24(SP) saved R0-R5
;    28(SP)   saved PSL
;    32(SP)   saved PC
;
;EM_INTERRUPT:
    MOVL   @(SP)+,R4
    MOVQ   (R4),R4
    CLRW   EM_STATUS1(R4)
    POPR   #^M<R0,R1,R2,R3,R4,R5>
    REI
;
    .PAGE
    .SBTTL SECOND INTERRUPT SERVICE ROUTINE
;
;EM_INTERRUPT2:
;    This interrupt routine service interrupts requested on the
; second interrupt line. This will signal an end of transfer period

```

```
; and the EMR should be shut off.  
;  
; Inputs:  
;   Same as EM_INTERRUPT:  
;  
EM_INTERRUPT2:  
    MOVL   @(SP)+,R3  
    MOVQ   (R3),R4  
    CLRW   EM_STATUS1(R4)  
    CLRW   EM_STATUS2(R4) ;shut off EMR 760  
    BBC    #UCB$V_INT,- ;not needed but looks good  
           UCB$W_STS(R5),-  
           DISMISS_INT2  
DISMISS_INT2:  
    MOVQ   UCB$L_FR3(R5),R3 ;restore R3 and R4  
    JSB    @UCB$L_FPC(R5) ;call the start I/O routine where  
           ;it was suspended  
    POPR   #^M<R0,R1,R2,R3,R4,R5> ;restore the stack  
    REI  
    .PAGE  
    .SBTTL END OF DRIVER  
;  
EM_END:  
    .END   ;End of driver label
```

Appendix B: User Program

```
; The purpose of this program is to monitor
; the DMA operation of the EMR 760. This program
; will initialize all buffer areas. It will create
; a file called DBAO:[GENE]ACQ.DAT;1 . The file
; will be used to transfer data from computer memory.
; The operator is assumed to be at a terminal so he
; too can provide control over the DMA operation.
; After everything is initialized, this program will
; send a message to the terminal. The operator will
; respond by turning the EMR 760 on. The entire
; operation is complete when the EMR 760 is shut off.
; To poll the EMR 760 device registers the program
; had to change the access mode from user to kernel.
; The program also raises priority to 16 (real-time
; application). The program also increases the
; working set to 300 pages, locks the pages in memory,
; and disables the swap mode.
; Three privileges are needed to execute this program.
; They are PSWAPM, ALTPRI, and LOGIO.
; This program takes complete control of the machine
; while polling the device registers, which is most
; of the time.
    .TITLE DATA ACQUISITION
    .IDENT /V01/
;
; OFFSETS
;
    $SSDEF      ;CONDITION CODE VALUES
    $IODEF      ;I/O FUNCTION VALUES AND OFFSETS
    $FIBDEF      ;FILE IDENTIFICATION BLOCK OFFSETS
    $RMSDEF      ;OFFSETS FOR RMS
;
; CONSTANTS
;
ACQ_START=.
MAX_BLOCK_SIZE=20000
DD_EFN=2
BUFFER_SIZE=32768
BYTE_COUNT=65535
VBN_OFFSET=64
EM_CSR1_BASE=^X80015A5A
EM_CSR2_BASE=^X80015A5C
EM_WC_CTR=^X80015A4E
;
; MACROS
;
    .MACRO DESCRIPTOR TEXT,?LABEL1,?LABEL2
    .LONG LABEL2-LABEL1
    .LONG LABEL1
    LABEL1: .ASCII /TEXT/
    LABEL2:
```

```

.ENDM DESCRIPTOR
.MACRO DISK_SLOW
$OUTPUT CHAN=TTCHAN,-
    BUFFER=TOO_FAST,-
    LENGTH=TOO_FAST_LEN
$CANCEL_S CHAN=EMCHAN
$DASSGN_S CHAN=EMCHAN
RET
.ENDM DISK_SLOW
;
.MACRO CHECK_FOR_ERR ARG1,?LA1
BLBS    R0,LA1
MOVW    ARG1,ERR_NUM
BRW    ERROR
LA1:
.ENDM CHECK_FOR_ERR
;
.MACRO CHECK_IOSB ?LA1
BLBS    R0,LA1
$FAO_S  CTRSTR=BLK_STRING,-
        OUTLEN=FAOLEN,-
        OUTBUF=FAODESCR,-
        P1=R5
$OUTPUT CHAN=TTCHAN,-
    BUFFER=FAOBUF,-
    LENGTH=FAOLEN
LA1:
.ENDM CHECK_IOSB
; INITIAL LIMIT FROM WSDEFAULT
; MAXIMUM LIMIT SET BY WSQUOTA
.MACRO INCREASE_WS
$ADJWSL_S PAGCNT=#300
CHECK_FOR_ERR #^X3130
.ENDM INCREASE_WS
;PRIVILEGE NEEDED PSWAPM
.MACRO LOCK_PAGES
$LCKPAG_S INADR=PAGES_ADDR
CHECK_FOR_ERR #^X3230
.ENDM LOCK_PAGES
;PRIVILEGE NEEDED PSWAPM
.MACRO SET_SWAP_MODE
$SETSWM_S SWPFLG=#1
CHECK_FOR_ERR #^X3330
.ENDM SET_SWAP_MODE
;PRIVILEGE NEEDED ALTPRI
.MACRO SET_PRIORITY ARG1
$SETPRI_S PRI=ARG1
CHECK_FOR_ERR #^X3430
.ENDM SET_PRIORITY
;
;
.LIST MEB
.PAGE
.SBTTL BUFFER AREAS
;

```

```

; BUFFER AREAS
;
; .PSECT BUFFER,NOEXE,WRT,BYTE
BUFFER1_ADDR: .BLKB 32768
BUFFER2_ADDR: .BLKB 32768
;
; .PAGE
; .SBTTL DATA STORAGE AREA
;
;
; .PSECT DATA,NOEXE,WRT,LONG
FAB_BLOCK:
$FAB    ALQ=MAX_BLOCK_SIZE,-
        FNM=<DBAO:[GENE]ACQ.DAT;1>,-
        FOP=<CTG,CIF,SUP,UFO>,-
        MRS=512,-
        ORG=SEQ,-
        RFM=FIX
;
; FAB BLOCK IS NEEDED TO CREATE A FILE IN RMS
EMNAME: DESCRIPTOR <_EMA0> ;NAME OF THE EMR DEVICE
EMCHAN: .BLKW 1 ;WORD TO RECEIVE THE CHANNEL NUMBER
DD_IOSB: .LONG      1 ;I/O STATUS FOR DISK DEVICE: INITIALIZE TO
        .LONG 0 ;SS$_NORMAL STATUS
TTNAME:   DESCRIPTOR <TT> ;TERMINAL TO BE USED TO OUTPUT ERROR MESSAGES
TTCHAN:   .BLKW 1 ;WORD TO RECEIVE TERMINAL CHANNEL NUMBER
; .PSECT DESCRIPTORS,EXE,WRT,LONG
DD_DESCR:
$QIO    EFN=\DD_EFN,-
        IOSB=DD_IOSB,-
        P1=BUFFER1_ADDR,-
        P2=\BUFFER_SIZE
        ;THE QIO MACRO GENERATES AN ARGUMENT
        ;LIST THAT CAN BE USED BY A CALL TO THE
        ;QIO SYSTEM SERVICE ROUTINE
EFN_STATE:
        .BLKL 1
EFN_DESCR:
$READEF EFN=\DD_EFN,-
        STATE=EFN_STATE
        ;THIS DESCRIPTOR WILL BE USED TO
        ;TEST THE EVENT FLAGS
TOO_FAST:
        .ASCII / EMR IS TOO FAST FOR THE DISK/
TOO_FAST_LEN=-.TOO_FAST
START_EMR:
        .ASCII /PROGRAM IS READY TO ACCEPT/
        .ASCII /INPUT FROM THE EMR 760/
START_EMR_LEN=-.START_EMR
FAODESCR:
        .LONG 80
        .LONG FAOBUF
FAOBUF:   .BLKB 80
FAOLEN:   .BLKW 1

```

```

.BLKW 1
BLK_STRING:
  DESCRIPTOR <AN ERROR OCCURRED WHILE TRANSFERRING BLOCK # !UL>
SUCCESS_STR:
  DESCRIPTOR < SUCCESSFUL COMPLETION !UL (D) BYTES TRANSFERRED>
PAGES_ADDR:
  .LONG ACQ_START
  .LONG ACQ_END
POLL_LIST:
  .LONG 3
  .LONG EM_CSR1_BASE
  .LONG EM_CSR2_BASE
  .LONG EM_WC_CTR
BLOCK_COUNTER:
  .LONG 0
WORD_COUNTER:
  .BLKL 1
;

; .PSECT DATA,NOEXE,WRT,LONG
; ERROR MESSAGES THAT HELP THE USER WHEN SOMETHING GOES AWRY!
; ERROR MESSAGES HAVE THE FOLLOWING FORMAT:
;

;   ERROR #NN GENERATED BY PROGRAM
; WHERE NN IS A NUMBER BETWEEN 0 AND 15
; THE FOLLOWING IS A LIST OF WHAT THE ERROR NUMBER MEANS:
;

; 00  -problems with the set event flag system service
; 01  -problems transferring data to disk
; 02  -problems locking pages in memory
; 03  -problems setting swap mode
; 04  -problems setting priority
; 05  -problems assigning EMR channel
; 06  -problems creating file: DBA0:[GENE]ACQ.DAT;1
; 07  -problems in QIO to EMR
; 08  -problems outputting to operator terminal
; 09  -problems reading event flag (Block 1)
; 10  -problems in QIO to disk (Block 1)
; 11  -problems reading event flag (Block 2)
; 12  -problems in QIO to disk (Block 2)
; 13  -problems waiting for single event flag
; 14  -problems in last QIO to disk
; 15  -problems with the FAO system service
; 16  -problems outputting byte count to terminal
;
;+ ++
ERR_BUF: .ASCII /ERROR #/
ERR_NUM: .ASCII /  GENERATED IN PROGRAM/
ERR_LEN: .LONG ERR_LEN-ERR_BUF
;

```

```

CREATE_STATUS:
    .BLKL 2      ; STATUS BLOCK USED WHEN CREATING THE FILE
;
;
; END OF VARIABLE ASSIGNMENTS AND STORAGE AREAS
;
;+ ++
;
    .PAGE
    .SBTTL PROGRAM ACQUISITION
;
; PROGRAM ACQUISITION
;
    .PSECT CODE,NOWRT,EXE,LONG
ACQUISITION:
    .WORD 0
    $ASSIGN_S CHAN=TTCHAN,-
        DEVNAM=TTNAME
    BLBS    R0,ALLOCATE_RES
    RET
;
ALLOCATE_RES:
    INCREASE_WS
    LOCK_PAGES
    SET_SWAP_MODE
    SET_PRIORITY #16
    $SETEF_S EFN=#DD_EFN
    CHECK_FOR_ERR #^X3030
    $ASSIGN_S CHAN=EMCHAN,-
        DEVNAM=EMNAME
    CHECK_FOR_ERR #^X3530
;
CREATE_FILE:
    $CREATE FAB=FAB_BLOCK
    CHECK_FOR_ERR #^X3630
    MOVW FAB_BLOCK+FAB$L_STV,-
        DD_DESCR+QIO$_CHAN
;
START_THE_EMR:
;PRIVILEGE NEEDED LOG_IO
    $QIO_S CHAN=EMCHAN,-
        FUNC=#IOS_READLBLK,-
        P1=BUFFER1_ADDR,-
        P2=#BYTE_COUNT
    CHECK_FOR_ERR #^X3730
    $OUTPUT CHAN=TTCHAN,-
        BUFFER=START_EMR,-
        LENGTH=#START_EMR_LEN
    CHECK_FOR_ERR #^X3830
;
POLL_CSRS_1:
    $CMKRNL_S POLL1,POLL_LIST
    BLBC    R0,CHECK_DISK2
    BRW    LAST_TRANSFER
;
CHECK_DISK2:
    $READEF_G EFN_DESCR
    CHECK_FOR_ERR #^X3930
    BBS    #DD_EFN,EFN_STATE,START_DISK1
    DISK_SLOW
;
START_DISK1:

```

```

$QIO_G DD_DESCR
CHECK_FOR_ERR #^X3031
SET_UP_FOR_2:
    INCL    BLOCK_COUNTER
    ADDL2  #VBN_OFFSET,-
           DD_DESCR+QIO$_P3
    MOVAL   BUFFER2_ADDR,-
           DD_DESCR+QIO$_P1
    MOVZWL  DD_IOSB,R0
    CHECK_IOSB
POLL_CSRS_2:
    $CMKRNL_S POLL2,POLL_LIST
    BLBC    R0,CHECK_DISK1
    BRW     LAST_TRANSFER
CHECK_DISK1:
    $READEF_G EFN_DESCR
    CHECK_FOR_ERR #^X3131
    BBS     #DD_EFN,EFN_STATE,START_DISK2
    DISK_SLOW
START_DISK2:
    $QIO_G DD_DESCR
    CHECK_FOR_ERR #^X3231
SET_UP_FOR_1:
    INCL    BLOCK_COUNTER
    ADDL2  #VBN_OFFSET,-
           DD_DESCR+QIO$_P3
    MOVAL   BUFFER1_ADDR,-
           DD_DESCR+QIO$_P1
    MOVZWL  DD_IOSB,R0
    CHECK_IOSB
    BRW     POLL_CSRS_1
LAST_TRANSFER:
    $WAITFTR_S EFN=#DD_EFN
    CHECK_FOR_ERR #^X3331
    MOVZWL  DD_IOSB,R0
    CHECK_IOSB
    $QIOW_G DD_DESCR
    CHECK_FOR_ERR #^X3431
    MULL3   #BUFFER_SIZE,BLOCK_COUNTER,R5
    ADDL2   #BUFFER_SIZE,R5
    SUBL2   WORD_COUNTER,R5
    $FAO_S  CTRSTR=SUCCESS_STR,-
           OUTBUF=FAODESCR,-
           OUTLEN=FAOLEN,-
           P1=R5
    CHECK_FOR_ERR #^X3531
    $OUTPUT  CHAN=TTCHAN,-
           BUFFER=FAOBUF,-
           LENGTH=FAOLEN
    CHECK_FOR_ERR #^X3631
DEALLOCATE:
    $DASSGN_S CHAN=EMCHAN
    $DASSGN_S CHAN=TTCHAN
    $DASSGN_S CHAN=DD_DESCR+QIO$_CHAN
    RET

```

```

POLL1:
    .WORD ^XFC
    MOVL #0,R0
    MOVL 4(AP),R2
    MOVL 8(AP),R3
LOOP1:
    BITW #1,(R2)
    BEQL DONE1
    BITW #1,(R3)
    BEQL LOOP1
    RET
DONE1:
    MOVL 12(AP),R4
    MOVZWL (R4),R5
    INCL R5
    MULL2 #2,R5
    MOVL R5,DD_DESCR+QIO$_P2
    MOVL R5,WORD_COUNTER
    MOVL #1,R0
    RET
POLL2:
    .WORD ^XFC
    MOVL #0,R0
    MOVL 4(AP),R2
    MOVL 8(AP),R3
LOOP2:
    BITW #1,(R2)
    BEQL DONE2
    BITW #1,(R3)
    BNEQ LOOP2
    RET
DONE2:
    MOVL 12(AP),R4
    MOVZWL (R4),R5
    INCL R5
    MULL2 #2,R5
    MOVL R5,DD_DESCR+QIO$_P2
    MOVL R5,WORD_COUNTER
    MOVL #1,R0
    RET
ERROR:    PUSHL R0
$OUTPUT CHAN=TTCHAN,-
          BUFFER=ERR_BUF,-
          LENGTH=ERR_LEN
          ; OUTPUT THE ERROR BUFFER TO THE TERMINAL DEVICE
$DASSGN_S CHAN=TTCHAN
POPL R0      ;GET STATUS CONDITION
RET
ACQ END=.    ;
.END ACQUISITION

```

Appendix C: Figures 8 and 9

Control Register 1														
	W	15	14	12	8	7	4	3	0					
STR	BURST	LNG	AB	0VI	IE2	IE1	XST	XSC	XPL	CYC	BLK	CYC	I/O	ON
BSY														
	R													

Bit(s)

0 ON-When set, BDC is "armed" for all DMA modes of operation. All other registers must be set prior to ON bit being set. On bit may be cleared as follows:

- * automatically, at end of block, in noncyclic operation
- * by the processor
- * by the RS bit of CR2
- * by the bus initialization (INIT) signal

1 I/O-(Input/Output) control for non-ESA DMA operation

- 1 =Input mode
- 0 =Output mode

2 CYC-(cyclic) mode control. When set, DMA transfers continue unrestricted by the Model 760 BDC. Cycling successive memory blocks defined by value in CYC BLK field of CR1. When cleared, DMA transfers proceed to end of current block, followed by channel turnoff.

3-4 CYC BLK(number of cyclic blocks). Two bit binary number from 0-3 specifies number of successive blocks 1-4. In cyclic mode, after completion of specified number of blocks, transfers proceed with the first block.

5 XPL-(External Party Line control). This bit determines external start line selection and input port DA or DB selection. In conjunction with CPL (control party line), input port DB is selected when XPL is set and CPL is a high-level. All other states define input port DA.

6 XSC-(External Start Control). When set, this bit enables the second line of a pair of external start input lines from either input port.

Figure 8. Control Register 1 (Sheet 1 of 2)
(Ref 4: Figure 1-2)

7 XST-(External Start). When set, DMA operation is inhibited even though 760 BDC is armed (ON bit set). XST can be cleared by external control line(s) associated with input ports DA, DB, or by the processor. Clearing is subject to the state of the XPL and XSC bits of CR1. The logic that clears XST may be expressed as follows:

XSC cleared: $(XSTA1)(/XPL)+(XSTB)(XPL)$
 XSC set: $(XSTA1)(XSTA2)(/XPL)+(XSTB1)(XSTB2)(XPL)$

8-9 IE1-2-(Interrupt Enable Bits). When set IE1 allows a priority interrupt at end of block (BLK END) during DMA operation. When set IE2 allows a priority interrupt to be externally initiated by user, or assigned to FIFO overflow signal. IE1-2 are initially cleared.

10 OVI-(Overflow Interrupt). When set, Interrupt 2 is assigned to FIFO overflow signal. When cleared external input signal (INT) may be assigned to Interrupt 2.

11 AB-(Adaptive Burst). When set, 760 BDC maintains burst mode until FIFO is flushed.

12-14 BURST LNG-(Burst Length). Value entered used to specify the number of data words to be collected in the FIFO prior to being transferred to memory in burst mode. Where 1=2, 2=4, ..., 7=14.

15 STR-((Strobe, write mode) INSTRB for data output register. Can be set or cleared directly by the processor, but only if INH bit of CR2 is set.

BSY-(Output Busy, read mode). When set indicates DOR is not ready for programmed controlled output. Any attempt to load DOR will result in no operation.

Figure 8. Control Register 1 (Sheet 2 of 2)
 (Ref 4: Figure 1-2)

15	DL4	DL3	INH	ESA	ADR	INC	SI2	SI1	DL2	DL1	T4	T3	T2	T1	INC	RS
ST4	ST3						MA17	MA16	ST2	ST1	BURST	CTR		BLK	CTR	

Control Register 2

Bit(s)

0 RS-(Reset, write only). When set by processor, cause channel initialization functionally equivalent to UNIBUS INIT. No read-modify-write instructions may be applied to this bit (or any other bit) with dual read-only, write only definitions.

1 INC-(Increment block counter, write only). Increments block counter (from current number in range 0-3). New number flushes the corresponding memory address register (MA_n - MAX_n) into MA CTR and contents of WC_n into WC CTR. Function is applicable to test and verification of block parameters. For all DMA operations, including ESA mode, MA1-MAX1 is automatically flushed to MA0-17 when ON bit is set.

0-1 BLK CTR-(Block Counter, read only). Value of count, in range 0-3, indicates cyclic block through which DMA transfers are being conducted. Increments automatically at time of changeover to new block. Does not increment id CYC bit or ON bit (CR1) are cleared during current block.

2-5 Test BITS 1-4-(write only). Apply only to automatic testing of 760 BDC (not defined herein). Bits are initially cleared.

BURST CTR -(read only). Value indicates the number of data entries in FIFO when burst mode is specified. Applicable to automatic testing only.

6-7 DL1-2-(Discrete levels outputs, write only 1 and 2). General purpose discrete level outputs for external user.

ST1-2-(Discrete input levels [status] 1-2, read only). General purpose discrete levels from external source.

Figure 9. Control Register 2 (Sheet 1 of 2)
(Ref 4: Figure 1-3)

8-9 SI1-2-(Stimulate interrupts 1 and 2, write only). When activated these bits cause priority interrupt generation independent of IE1 and 2 (CR1).

MA16-17-(Memory Address bits 16 and 17, read only). Two highest order bits of MA counter.

10-11 ADR INC-(Address Increment). Used to specify means by which MA counter is to be incremented during non-ESA DMA modes. Bits are initially cleared.

 00 =word addressing
 01 =alternate word addressing
 10 =byte addressing
 11 =not applicable

12 ESA-(External Specified Addressing). When set memory addressing is externally specified. Bit is initially cleared.

13 INH-(Inhibit data output register control lines). When set, inhibits INSTRB-INRQ control circuit, allowing processor control of DOR. Bit is initially cleared.

14-15 DL3-4 (Discrete output levels, write only). General purpose discrete level outputs for external control.

ST3-4 (Discrete input levels [status] 3-4, read only). General purpose discrete level inputs from external user interface. Bits are initially cleared.

Figure 9. Control Register 2 (Sheet 2 of 2)
(Ref 4: Figure 1-3)

Vita

David L. Rall was born in Vincennes, Indiana on August 24, 1957. His parents are Mr and Mrs William J. Rall. He graduated from Coronado High School in 1975 and subsequently attended Northern Arizona University. He graduated with a Bachelor of Science in Computer Science and was commissioned in the Air Force through the ROTC program on May 19, 1979. His first assignment in the Air Force was to attend the Master's degree program in Computer Systems offered by the Electrical Engineering department at the Air Force Institute of Technology

Permanent Mailing Address:

227 E. Taylor
Tempe, Arizona 85281

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE			READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/300-13	2. GOVT ACCESSION NO. AD-A100 777	3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) WRITING A SOFTWARE PACKAGE FOR AN EMR 760 BDC ON THE VAX-11/730		5. TYPE OF REPORT & PERIOD COVERED MS Thesis	
7. AUTHOR/s:		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS Data Conversion Branch Air Force Weapons Laboratory (AFWL) Kirtland AFB, New Mexico		12. REPORT DATE December, 1980	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		13. NUMBER OF PAGES 125	
		15. SECURITY CLASS. (of this report) Unclassified	
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES Approved for public release: IAW AFR 190-17 FREDERIC C. LYNCH, Major, USAF Director of Public Affairs			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) I/O Drivers Bufferred Data Channel VAX-11/780 I/O Devices			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A device driver for an EMR 760 BDC was written for the VAX-11/780. An Application program was written to use the driver and transfer rates in excess of 100 000 sixteen bit words per second were realized. The package successfully transferred 10 Mb to disk using DMA operations. The techniques for writing device drivers on the VAX-11/730 are discussed.			

END

DATE
FILMED

7 - 81

DTIC